

図解

16ビットマイクロコンピュータ

# 80286

の使い方

本岡 善剛 著 ● オーム社



図解 16ビットマイクロコンピュータ  
**8086 の使い方**

(A 5判 190頁)

井出裕巳 著

16ビットマイクロコンピュータの中で最も多く使われているインテル社の8086について、概要から各種命令、動作、プログラミングなどのポイントを、具体的かつ平易に解説しました。

図解 16ビットマイクロコンピュータ  
**MC68000 の使い方**

(A 5判 256頁)

小島 進 著

16ビット CPU の一つである MC 68000について、ソフトウェア、ハードウェア、システムの構成、各種ファミリー、開発ツールなどを図解によるページユニット方式で、具体的かつ平易に解説。

図解 マイコンの基礎知識

(A 5判 250頁)

矢田光治 著

マイコンの基礎知識を94項目ページ単位に要約、2色刷で視覚的に解説しました。初心者にはマイコン事典として、専門家にはポイントの整理に役立ちます。

制御用マイコンの  
**作り方・使い方**

(B 5判 240頁)

北川一雄 著

本書は、8085、Z-80系のマイコンを用いた各種制御回路・装置の作り方と、制御用マイコンの使用例を基礎と応用に分け、具体的プログラム例を示しました。





図解

16ビットマイクロコンピュータ

# 80286

## の使い方

本岡善剛 著

オーム社

UNIX は AT & T 社ベル研究所の商標.

CP/M-86 は米国デジタルリサーチ社の商標.

MS-DOS, XENIX は米国マイクロソフト社の商標.

I<sup>2</sup>ICE は米国インテル社の商標.

本書は、「著作権法」によって、著作権等の権利が保護されている著作物です.

本書の全部または一部につき、無断で次に示す〔 〕内のような使い方をされると、著作権等の権利侵害となる場合がありますので御注意ください.

〔 転載、複写機等による複写複製、電子的装置への入力等〕

学校・企業・団体等において、上記のような使い方をされる場合には特に御注意ください.

お問合せは下記へお願いします.

〒101 東京都千代田区神田錦町 3 - 1 Tel. 03-233-0641

株式会社 オーム社出版局（著作権担当）

## は し が き

初期のマイクロプロセッサは、その機能を限定することによって半導体集積回路にコンピュータの CPU（中央処理装置）を実現したものであった。したがって、その大きさにおいても能力においても小さなコンピュータを意味した。しかし、半導体技術の発達によって、現在のマイクロプロセッサは決して小さなコンピュータではなく、その能力は 10 年ほど前の大型汎用コンピュータと同レベルかそれ以上のものになっている。

しかし、マイクロプロセッサの重要な使命は特別な権利をもつ技術者に限らず、だれに対しても広くそのコンピュータの能力を提供することである。このことはまたマイクロプロセッサの大きな魅力でもある。このためマイクロプロセッサのアーキテクチャは、ミニコン、大型汎用コンピュータの技術を学ぶ一方で、それをだれもが利用できるような形に昇華されてきた。この点においてインテル社の 8086 は最も成功したマイクロプロセッサの 1 つであろう。

本書で述べる 80286 は、8086 の上位に当たるマイクロプロセッサであるが、この 2 つのマイクロプロセッサの間には単なる改良型というものの以上の大きな違いがある。このためインテル社は、80286 以後のマイクロプロセッサを 8086 などに対して第 3 世代のマイクロプロセッサと呼んでいる。

80286 はマルチタスクオペレーティングシステムを実行することを念頭において設計されたマイクロプロセッサである。この意味で 80286 は 16 ビットのマイクロプロセッサであるが、32 ビットのマイクロプロセッサ 80386 と同レベルに位置する。80286 は、マルチタスクオペレーティングシステムを設計するうえで重要な処理をマイクロプロセッサのハードウェアとマイクロコードで処理する。

80286 は高速の 8086 としても利用できるが、本書ではマルチタスクオペレーティングシステムを設計するうえで重要な 80286 の基本的機能と命令を中心にまとめた。8086 と共通な命令の詳細な解説は他の 8086 関係のすぐれた各著書にまかせることにするが、本書自体は 8086 を知らなくても十分に読めるように解説

は し が き

した。

本書を執筆するに当たっては多くの方々のお世話になった。これらの方々の協力なしには本書は完成しなかった。この場を借りて心から謝意を表します。インテルジャパン株式会社トレーニングセンター前マネージャー大野民生氏は、本書の執筆を私に勧めてくださった。同社のマーコムマネージャー松本芳夫氏はインテル社のユーザーズガイド、データシートを利用する上で協力的に対応していただいた。また、同社トレーニングセンターのスタッフ全員からは、執筆において絶え間のない励ましをいただいた。特に池田聡氏は私の読みにくい原稿をていねいに読み、多くの注意点を指摘していただいた。

本書にあげたプログラムはすべてインテル社 MDS シリーズⅣと I<sup>2</sup>ICE また SIM 286 を使用して動作を確認したものであり、その他の内容についても正確さには細心の注意を払った。しかし、なお記述中に誤りが残っているとすれば、それはすべて著者の責任である。

最後に、技術的知識とそれらを本にまとめるセンスとは別のものであり、私の原稿がこのような本にまとめることができたことはオーム社出版部の方々の努力に負うところが大きい。心から感謝の意を表します。

1987 年 3 月 12 日

著者しるす

# 目 次

## 1. 80286 の概要

1-1	8086 から 80286 へ	2
1-2	マイクロコンピュータの構成	5
1-3	バスサイクル	8
1-4	レジスタ構成	11
1-5	セグメンテーション	13
1-6	リアルモードとプロテクトモード	17

## 2. リアルモードでの使用

2-1	メモリ、I/O 参照と転送命令	22
2-2	演 算 命 令	27
2-3	制 御 命 令	30
2-4	ストリング命令	35
2-5	拡 張 命 令	40

## 3. プロテクトモードでの使用

3-1	セグメントキャッシュ	46
3-2	ディスクリプタテーブル	49
3-3	セグメントレジスタの保護	55
3-4	メモリ参照の保護	60
3-5	仮 想 記 憶	62

## 目 次

3-6	ディスクリプタテーブルの扱い	65
3-7	プロテクトモードの初期設定	67
4. 特権レベルによる保護		
4-1	特 権 レ ベ ル	72
4-2	データセグメント、スタックセグメントの特権保護	76
4-3	コードセグメントの特権保護	79
4-4	コールゲートによる制御移行	81
4-5	スタックセグメントの保護	84
4-6	RET 命令による制御移行	86
4-7	コンフォーミングコードセグメント	88
4-8	「トロイの木馬」問題	90
4-9	I/O 参照の保護	96
5. 割り込み処理		
5-1	割り込みの原因	98
5-2	割り込みのプロセスと IDT	100
5-3	ハードウェア割り込み	105
5-4	ソフトウェア割り込み	107
5-5	内部割り込み	110
6. タスクとタスクスイッチ		
6-1	シングルタスク	112
6-2	マルチタスク	113
6-3	LDT と LDT ディスクリプタ	117
6-4	タスクの定義	119
6-5	タスクスイッチ	121
6-6	タスクゲート	125
6-7	タスクスイッチの例	127

## 7. 保護例外

7-1 保護例外	134
7-2 スタックエラー	136
7-3 TSSエラー	137
7-4 Pビットエラー	138
7-5 一般保護エラー	139
7-6 二重エラー	140
7-7 例外処理と再実行	141

## 8. 80286のハードウェア

8-1 CPUモジュールの構成	144
8-2 メモリインタフェース	155
8-3 I/Oインタフェース	160
8-4 ローカルバス制御	162
8-5 システムバス制御	163

## 9. 数値演算コプロセッサ 80287

9-1 80287のアーキテクチャ	166
9-2 データの表現	169
9-3 レジスタスタックの基本的使用	171
9-4 演算命令と関数命令	175
9-5 80286と80287の接続	179
9-6 例外処理	183
9-7 80287のためのサポート	184

## 10. プログラム開発

10-1 開発システム	188
10-2 システム開発とユーティリティプログラム	189

目	次	
10-3	デバッグ	192
付	録	
I.	セグメントおよびディスクリプタのまとめ	195
II.	80286 命令コード	198
参 考 文 献		222
お わ り に		223
索 引		225

# 1. 80286の概要

マイクロプロセッサ 80286 は、16 ビットのレジスタ群と 16 ビットのデータバスをもつプロセッサで、マイクロプロセッサ 8086 の応用プログラムを、ほとんど書き換えることなく実行することができるものである。80286 は、2 つの動作モードをもつ。リアルモードと呼ばれるモードでは、80286 はスピードの速い 8086 として動作する。しかし、他方のプロテクトモードと呼ばれるモードにおいて、80286 は本来の能力を発揮する。

### 1-1 8086 から 80286 へ

〔1〕 16ビットマイクロプロセッサの歴史 インテル社のマイクロプロセッサの歴史は表1・1に示すように、4ビットの4004において始まった。4004の目的は電卓の制御だったが、メモリに記録したプログラムを順次実行するストアードプログラム方式、または演算機能をもつことなどは、4004をコンピュータの中央処理装置(CPU)と呼ぶに十分な特徴であった。

8ビットの8080に至って、CP/Mなどのディスクオペレーティングシステム(DOS)も作られ、マイクロコンピュータの開発システム、またはパーソナルコンピュータにも使用された。しかし、8085はどちらかといえば、制御に使用されることが多く、かつての16ビットミニコンピュータと比較して、性能において大きな開きがあった。

しかし、16ビットの8086においては、部分的にはミニコンピュータの置き換えに利用されるまでの性能をもつに至った。8086では、CP/M-86、MS-DOSなどのDOSが普及し、8086上で走る応用プログラムは膨大な量になる。世界におけるその総額は1986年において60億ドルといわれている。

〔2〕 80286の必要性 図1・1(a)に示すようなシングルタスク、シングルユーザのシステムでは、8086は十分な能力をもつが、同図(b)、(c)に示すようなマルチタスクのシステムを8086を使用して実現することが困難な場合がある。ここで、タスクとは実行可能なプログラムのことである。マルチタスクシステムは、メモリに複数のプログラムを置き、図1・2に示すように、必要に応じてCPUの使用権を他のタスクに渡しながら処理を進めることができる。

マルチタスクシステムの応用としては、図1・1(b)に示すリアルタイムシステムを作ることができる。このようなシステムでは、ロボットがベルトコンベアに部品を置く作業をしているが、環境の変化に応じてリアルタイムに(すぐに)、次のプログラムを実行することができる。2つのタスクはまったく別のプログラムでもよいし、また、同じプログラムでもかまわない。

図1・1(c)に示すシステムでは、1つのコンピュータに2台以上の端末が接続され、複数のユーザがそれぞれの端末から1つのシステムを共有することができる。2人のユーザは別々のプログラムを実行してもよいし、また、同じエディタ

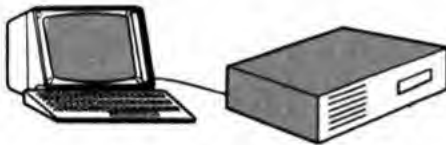
表 1・1 インテル社のマイクロプロセッサの発展

発表時期	マイクロプロセッサ	処理データの大きさ	実メモリ空間	I/O 空間
1971	4004	4 ビット	4 K バイト (ROM) 5 120 ビット (RPM)	16×4 ビット (入力ポート) 16×4 ビット (出力ポート)
1972	8008	8 ビット	16 K バイト	
1974	8080	8 ビット	64 K バイト	256 バイト
1976	8085	8 ビット	64 K バイト	256 バイト
1978	8086	16 ビット	1 M バイト	64 K バイト
1982	80186	16 ビット	1 M バイト	64 K バイト
1982	80286	16 ビット	16 M バイト	64 K バイト
1985	80386	32 ビット	4 G バイト	64 K バイト

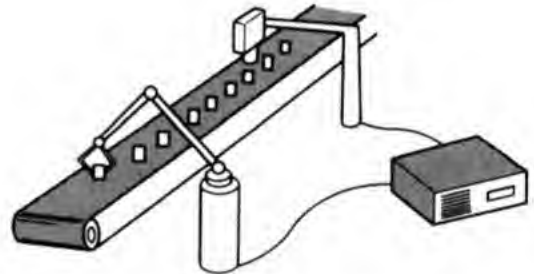
1 K バイトは 1024 バイト、

1 M バイトは 1024 K バイト、

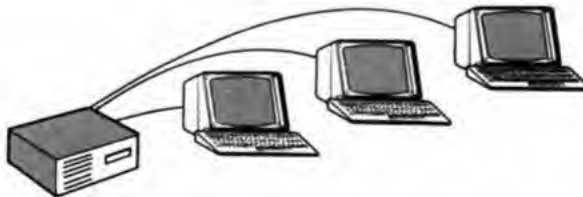
1 G バイトは 1024 M バイトをそれぞれ表す、



(a) シングルタスク、シングルユーザシステム



(b) リアルタイムシステム



(c) マルチタスク、マルチユーザシステム

図 1・1 コンピュータシステムの利用形態

必要に応じて CPU を別の仕事にまわす、

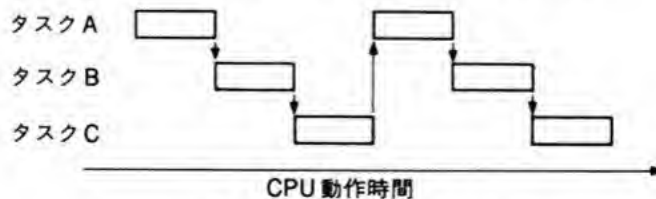


図 1・2 CPU 時間の分割使用

## 1 80286 の 概 要

(編集プログラム)を使用してユーザ独自のテキストを作成することもできる。このシステムでは、各ユーザが実行するプログラムはそれぞれタスクとなり、各タスクがCPUを**時分割**(**タイムスライシング**)で使用する。もちろん、CPUを時分割で使用することによって、図1・2に示すように、1つのタスクの実行時間は長くなる。しかし、ユーザが端末を使用して作業するとき、このような実行時間の変化はほとんど感じないだろう。各ユーザは他のユーザとは関わりなく、自分だけがシステムを独占しているように使用することができる。

ここで、マルチタスクシステムでは、タスクへのCPUの切り換え、メモリ管理、ファイル管理などの作業が必要である。たとえば、図1・1(c)のシステムにおいて、あるユーザが実行させたプログラムが暴走することによって、別のユーザのプログラムの実行をつぶすようなことがあってはならないし、ディスクに格納した大切なデータを他のユーザに読まれたり、書き換えられるようなことがあってはならない。これらはオペレーティングシステム(OS)の仕事であるが、CPUの能力によっては、OSのオーバヘッドばかりが大きくなって、本来のユーザプログラムに分け与えるCPU時間が少なくなってしまう。こうなると、ユーザはCPUを時分割で共有していることを感じざるをえず、実用にならない場合がある。

80286は、マルチタスクシステムに必要な、タスクスイッチ、メモリ管理、各種の保護機能をCPU自体がもち、8086の応用プログラムをマルチタスクで実行するためのマイクロプロセッサである。

## 1-2 マイクロコンピュータの構成

〔1〕 **マイクロコンピュータの基本構成** マイクロコンピュータは、基本的には図1・3に示すように、**CPU**、**メモリ**、**I/O** (Input/Output, 入出力装置) の3つの要素によって構成される。CPUは、演算などのコンピュータにおいて最も中心となる仕事をする部分である。また、その他にコンピュータ全体を能動的に制御するいくつかの機能をもつ。メモリは、プログラム、データを一時的に記録する装置で、CPUによってリード(メモリからCPU内部のレジスタへデータを読む動作)、ライト(CPU内部のレジスタからメモリへデータを書く動作)が実行される。I/Oには周辺装置との**インタフェース**を接続する。CPUとI/Oインタフェース間のデータのリード、ライトは、メモリと同様の方式で実行される。インタフェースは、データを外部の周辺装置に適合するように信号を変換する。

〔2〕 **バス** これらの3つの要素はバスと呼ばれる伝送路によって接続されている。バスは、**制御バス**、**アドレスバス**、**データバス**に分類される。制御バスは、「メモリからデータをリードする」、「メモリにデータをライトする」、「I/Oからデータをリードする」、「I/Oにデータをライトする」などの**コマンド信号**をCPUからメモリ、またはI/Oに送るために使用する。

たとえば、「メモリからデータをリードする」ことを表すコマンド信号をCPU

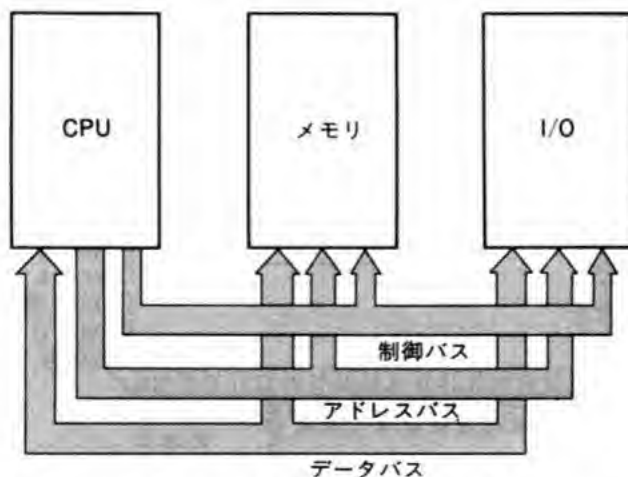
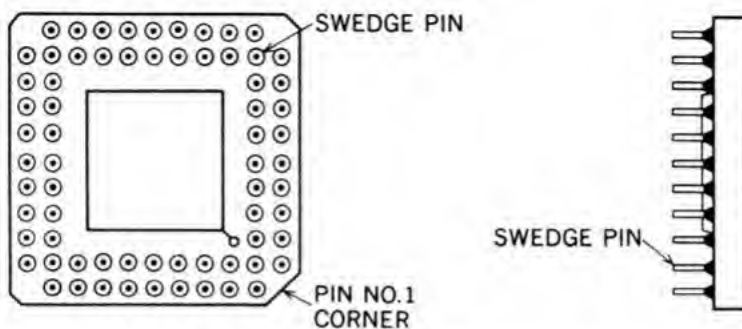
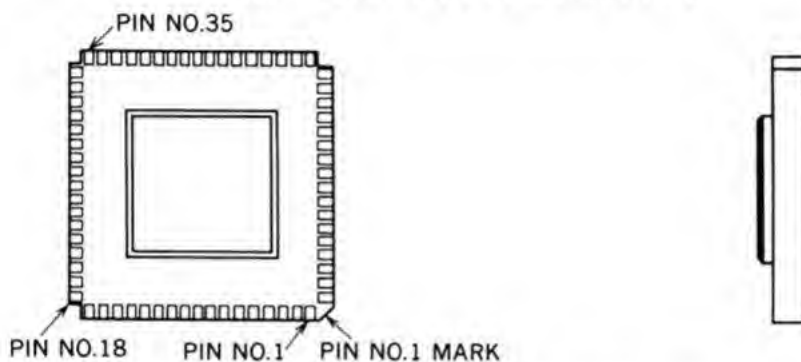


図 1・3 マイクロコンピュータの構成

# 1 80286 の 概 要



(a) PGA (ピングリッドアレイ) パッケージ



(b) JEDEC タイプ LCC (リードレスチップキャリア) パッケージ

図 1・4 80286 マイクロプロセッサ

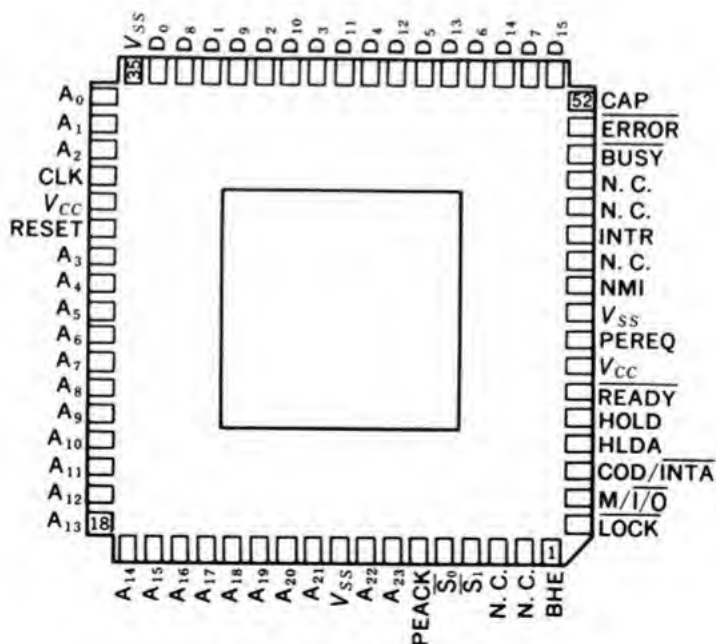


図 1・5 80286 端子配置図

からメモリに送ることによって、メモリがデータを出力するようにハードウェアを設計する。このとき、同時にアドレスバスを使用して、CPU からアドレスをメモリに送る。メモリは CPU から送られたコマンド信号とアドレスに応答して、対応するデータをデータバスへ出力する。その後、CPU はデータバス上の信号を入力し、内部のレジスタに記録する。データライトの場合は、逆に、CPU がデータをデータバスに出力し、メモリがデータバス上の信号を入力する。このようなデータのリード、ライト動作は、I/O の場合もメモリのときと同様に実行される。I/O へのデータのリード、ライトのときには、「I/O からデータをリードする」、または「I/O にデータをライトする」ことを表すコマンド信号が CPU から出力され、アドレスバスで指定された I/O インタフェースが応答する。

80286 は、図 1-4 に示すようにピングリッドアレイパッケージに入っているものと、リードレスチップキャリアパッケージに入っているものとがある。どちらのパッケージも 68 本の端子をもち、このうち 64 本の端子が使用されている。図 1-5 に、リードレスチップキャリアパッケージの場合の端子配置図を示す。

メモリのアドレス信号は  $A_0-A_{23}$  の 24 ビットの端子から出力される。I/O のアドレス信号は、 $A_0-A_{15}$  のアドレスバスの下位 16 ビットから出力される。メモリ、I/O はともに 1 バイト単位で 1 つのアドレスが与えられるから、80286 は  $2^{24}$  バイト (= 16 M バイト) の実メモリ空間と、 $2^{16}$  バイト (= 64 K バイト) の I/O 空間をもつ。

データの入出力には  $D_0-D_{15}$  の 16 ビットの端子が使用される。80286 はコマンド信号を直接出力する端子をもたない。そのかわりに、 $\overline{S}_0, \overline{S}_1$  の端子から出力する 2 ビットのステータス信号によって、80286 からメモリ、I/O に送るコマンドを表現する。80286 がメモリ、I/O に対して何の動作も実行しない状態 (アイドル状態) のときは、 $\overline{S}_0, \overline{S}_1$  はともに High になっている。また、メモリに対する動作か、I/O に対する動作かの区別は  $M/\overline{IO}$  端子の出力によって区別できる。 $M/\overline{IO} = \text{High}$  のときは、メモリに対する動作であることを表し、 $M/\overline{IO} = \text{Low}$  のときは、I/O に対する動作であることを表す。 $\overline{S}_0, \overline{S}_1, M/\overline{IO}$  の信号をデコードすれば、コマンド信号を作ることができる。

## 1-3 バスサイクル

〔1〕 **バスサイクルの構成** 80286 がメモリ、または I/O からデータのリード、ライトを実行する時間を**バスサイクル**と呼ぶ。80286 が、データリード、データライトを連続して実行する場合のバスサイクルのタイミングを、図 1・6 のタイミングチャートに示す。CLK (クロック) 信号は 80286 の CLK 端子から入力される信号で、**システムクロック**と呼ばれる。80286 は、内部でシステムクロックを 2 分周して、**プロセッサクロック (PCLK)**と呼ばれる信号を作る。80286 の内部動作は、プロセッサクロックに同期して実行されるが、80286 の信号のタイミングチャートは、システムクロックを基準にして記述されている。プロセッサクロックが 8 MHz で動作する 80286 には、2 倍の 16 MHz のシステムクロックを供給する。

図 1・6 に示すように、80286 は 2 つのプロセッサクロックで 1 回のバスサイクルを実行する。バスサイクルの中で、最初のプロセッサクロックを**ステータスサイクル**と呼び、記号  $T_s$  で表す。また、次のプロセッサクロックを**コマンドサイクル**と呼び、記号  $T_c$  で表す。 $T_s$  では、80286 はステータス信号  $\overline{S_0}$ 、 $\overline{S_1}$  を出力する。

〔2〕 **パイプラインドアドレッシング** アドレスと  $M/\overline{IO}$  は同じタイミングで出力されるが、これらの信号は少なくとも、 $T_s$  の 1 システムクロック以前に出力されている。アドレス、 $M/\overline{IO}$  は、 $T_c$  の  $\phi 1$  (フェーズ 1) の最後まで出力され続けることが保証されている。もし、ここで次のバスサイクルが発生したときは、80286 は現在のバスサイクルの実行が終了する、しないに関係なく、 $T_c$  の  $\phi 2$  から次のバスサイクルのアドレスと  $M/\overline{IO}$  を出力する。

したがって、たとえば図 1・6 のように、リードサイクルのすぐ後に次のライトサイクルが実行される場合は、リードサイクルの  $T_c$  の  $\phi 2$  において、リードサイクルの実行が完全に終了していないにもかかわらず、次のアドレスが出力される。

80286 では、データの入出力を実行中に次のバスサイクルのアドレスを出力するように、2 つの連続するバスサイクルの一部を重複させることができ、このことを**パイプラインドアドレッシング**と呼ぶ。パイプラインドアドレッシングの目

的は、メモリの応答スピードがシステムの性能へ影響することをできるだけ小さくすることである。

80286 のメモリ空間は 16 M バイトの大きさをもち、これは 8086 のメモリ空間の 16 倍の大きさとなる。また、バスサイクルの時間も、同じ 8 MHz のクロックで動作する 8086 と比較して 2 倍速くなっている。しかし、プロセッサの性能だけを上げて、メモリの応答スピードが遅ければ、システム全体のスピードの向上は望めない。では、応答スピードの速いメモリを用意すればよいということになるが、コストの問題がある。80286 では、パイプラインアドレッシングによって、比較的低速のメモリを使用した場合も、システム全体の性能をそれほど落とさないように設計することができる。

〔3〕 コマンドサイクルとウェイトサイクル リードサイクルにおいて、80286 は  $T_c$  の最後に、 $D_0-D_{15}$  の端子からデータを入力する。このとき、少なくとも  $T_c$  の最後の時間の前後、セットアップタイムとホールドタイムの間、データバス上のデータは安定していなければならない。ライトサイクルにおいては、80286 は  $T_s$  の  $\phi 2$  から、バスサイクルの 1 システムクロック後までの間、データを  $D_0-D_{15}$  の端子に出力する。

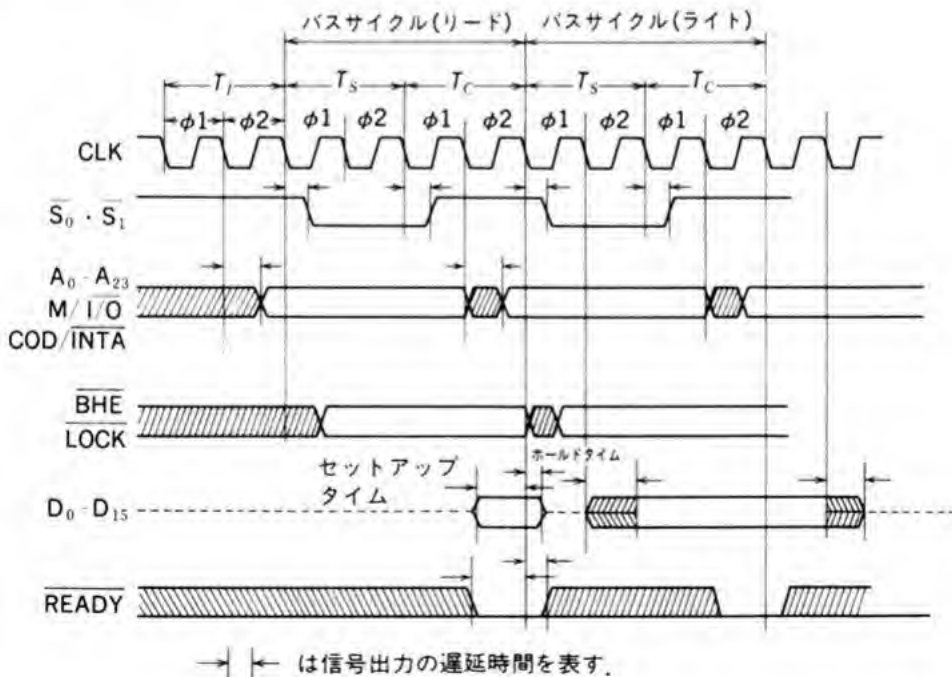


図 1・6 基本バスサイクル

## 1 80286 の 概 要

リードサイクルの場合、メモリは  $T_c$  の最後からセットアップタイム前までにデータを出力するように応答しなければならない。また、ライトサイクルの場合、メモリは  $T_c$  の最後から1システムサイクル後までにデータを入力するように応答しなければならない。もし、正しくデータのリード、ライトができないような応答スピードの遅いメモリを使用する場合は、 $T_c$  の後に任意の数の  $T_c$  を挿入して、バスサイクルを延長することができる。

このとき、データのリードはバスサイクル最後の  $T_c$  の終わりで実行される。また、ライトサイクルにおいて、最後の  $T_c$  の1システムクロック後までデータを出力する。バスサイクルを延長するために挿入する  $T_c$  を**ウェイトサイクル**と呼ぶ。バスサイクルにウェイトサイクルを入れるか、入れないかは  $\overline{\text{READY}}$  (レディ) 信号によって制御できる。バスサイクルの最後に  $\overline{\text{READY}} = \text{Low}$  であれば、バスサイクルは終了するが、逆に、 $\overline{\text{READY}} = \text{High}$  のときは、 $T_c$  の後に、再び  $T_c$  が繰り返される。 $\overline{\text{READY}}$  を High のままにしておけば、ウェイトサイクルは繰り返され、バスサイクルは終了しない。

## 1-4 レジスタ構成

次に、80286の内部構成について考える。アセンブリ言語でプログラムする上で知っておくべきことは、80286がどのようなレジスタ構成をもつかということである。80286のレジスタ構成を図1・7に示す。また、**FLAG**、**MSW**のそれぞれのビット構成を図1・8(a)、(b)に示す。**汎用レジスタ**と呼ばれる一連のレジスタ群の種類および扱いは、まったく8086の汎用レジスタと同じである。

ワード(16ビット)汎用レジスタには、AX, BX, CX, DX, BP, SP, SI, DIの8本のレジスタがある。8086, 80286において、このように、汎用レジスタ名が統一されていないのは、ある種の命令では、自動的にレジスタの役目が決まっているからである。各レジスタは、その特別な役目を表す名前をもっている。たとえば、CXはカウンタレジスタを表す。これは、LOOP命令、ストリング命令において繰り返し処理の回数を指定するために使用される。

特定の命令において使用するレジスタを暗黙的に定義しておけば、命令コードを短くすることができる。逆に、レジスタの使用に自由がなくなるが、プログラムを書くとき、結局はプログラマがレジスタの使用目的を決めるのだから、レジスタ使用の自由度が小さいことはそれほどの欠点にはならない。

**セグメントレジスタ**の種類も8086のものと同じである。しかし、80286のセグメントレジスタには、それぞれ、48ビットの**セグメントキャッシュ**が付属している。セグメントキャッシュはセグメントの属性を記録し、80286がメモリ管理機能を実行するとき内部的に使用するレジスタであるが、セグメントキャッシュには直接に値を代入することはできない。

FLAGは図1・8(a)に示したようなビットに区分され、それぞれのビットが演算結果の状態を表したり、また、80286の動作を制御するために使用される。FLAGのビット構成はビット0からビット11までが、8086のFLAGと同じであるが、新たに**IOPL**と**NT**が追加されている。

**MSW**(マシンステータスワード)はFLAGと似た性格のレジスタであり、図1・8(b)に示すように、下位4ビットが使用されている。MSWはFLAGよりももっと、80286の動作に根本的に影響するフラグを表す。

# 1 80286 の 概 要

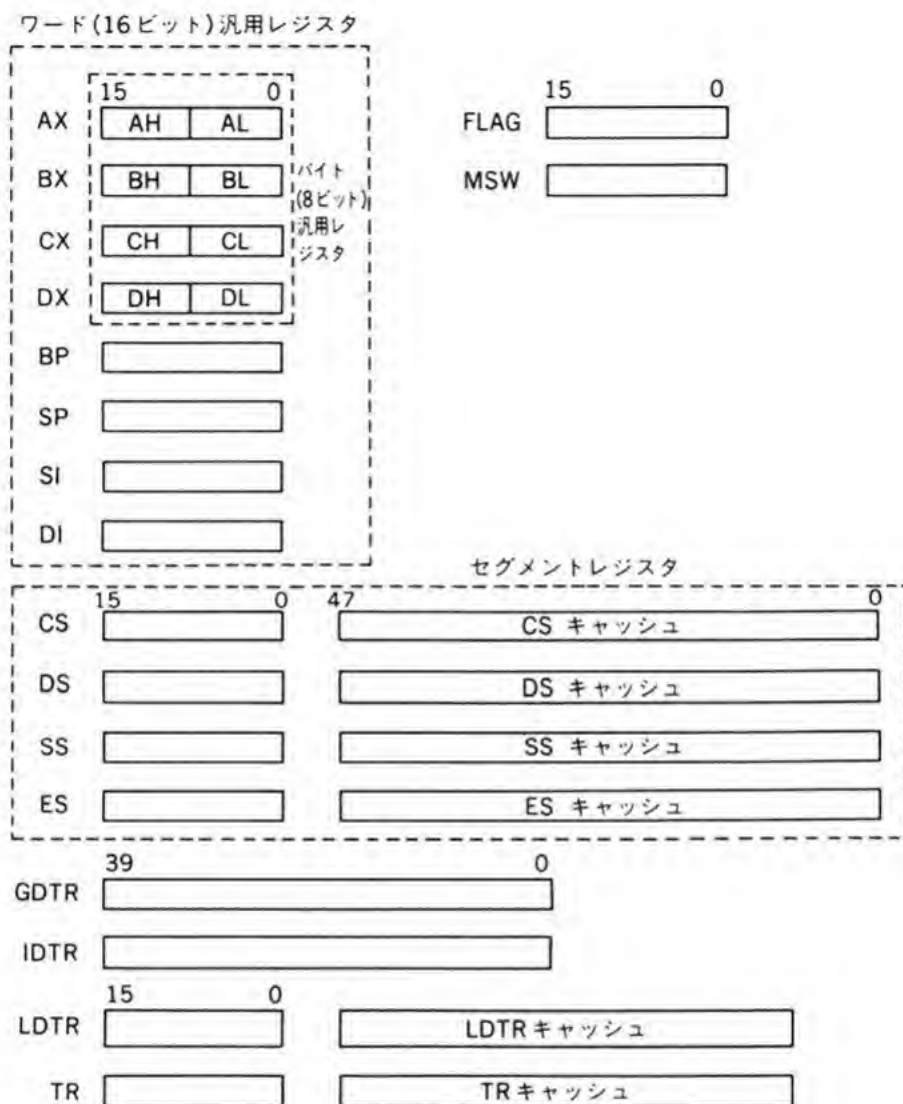


図 1・7 80286 のレジスタ構成

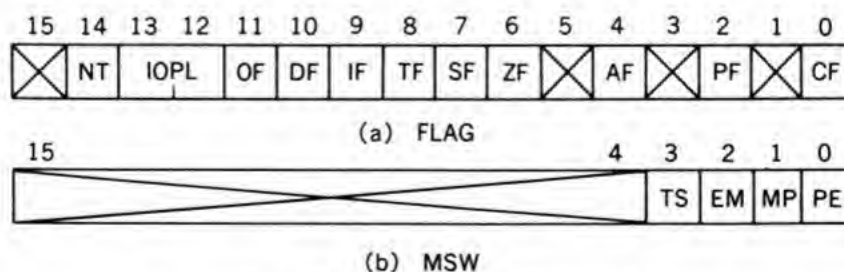


図 1・8 FLAG, MSW のビットフィールド

## 1-5 セグメンテーション

〔1〕 **セグメントと論理アドレス** 80286 のアドレスバスは 24 ビットであるから、 $2^{24} = 16 \text{ M}$  バイトのメモリ空間をもつ。(ただし、後に述べるように、リアルアドレスモードで使用するときは、アドレスバスの下位 20 ビットだけを使用するので、 $2^{20} = 1 \text{ M}$  バイトのメモリ空間をもつ)。80286 はメモリのリード、ライトにおいて端子  $A_0 - A_{23}$  に 24 ビットのアドレスを出力する。アドレスバスに出力される 24 ビットの信号を **物理アドレス** と呼ぶ。

しかし、80286 内部においては(すなわち、プログラムにおいては)物理アドレスを直接使用することはない。

プログラムは、命令を定義したコード領域、変数を定義したデータ領域、そしてスタック領域から構成され、図 1-9 に示すようにメモリに配置される。これらの領域は、それぞれ、メモリの連続した領域に定義されるが、領域相互間には必ずしも連続したメモリに配置する必要はない。

このように領域を相互に独立に扱うためには、メモリ空間内の領域の位置を表すアドレスと、領域内のデータの位置を表すアドレスの、2 つのアドレスによってメモリ空間内の特定データの位置を表現すればよい。

ここで、独立に扱うことのできる領域を**セグメント**と呼ぶ。メモリ空間内のセグメントの位置を表すアドレスを**セグメントセレクト**と呼ぶ。また、セグメント内の特定のデータの位置を表すアドレスを**オフセット (相対アドレス)**と呼ぶ。そして、セグメントを使用した 2 次元的なアドレス指定法を**セグメンテーション**と呼ぶ。セグメントセレクトとオフセットの組み合わせによってメモリ空間内の任意のバイト、ワードデータを一意に指定することができる。セグメントセレクトとオフセットを**論理アドレス**と呼ぶ。

〔2〕 **セグメントレジスタの使用** オフセットは、メモリを参照する命令の中で指定するが、セグメントセレクトは命令の中で直接、指定するのではなく、あらかじめセグメントレジスタに初期設定しておく。セグメントレジスタは図 1-10 に示すように、CS、SS、DS、ES の 4 種類があり、メモリ参照の種類によって使用されるセグメントレジスタが、表 1-2 のように決められている。80286 は命令をリードする(コードフェッチ)ときは、必ず CS によって指定されるセ

## 1 80286 の 概 要

グメントから命令をリードする。このとき、オフセットは必ず IP の値を使用する。PUSH, POP 操作によって、データを保存したり、または、保存したデータを読み出したりするときは、セグメントセクタに必ず SS が使用される。

また、オフセットは SP によって指定される。オフセットを表現するために、

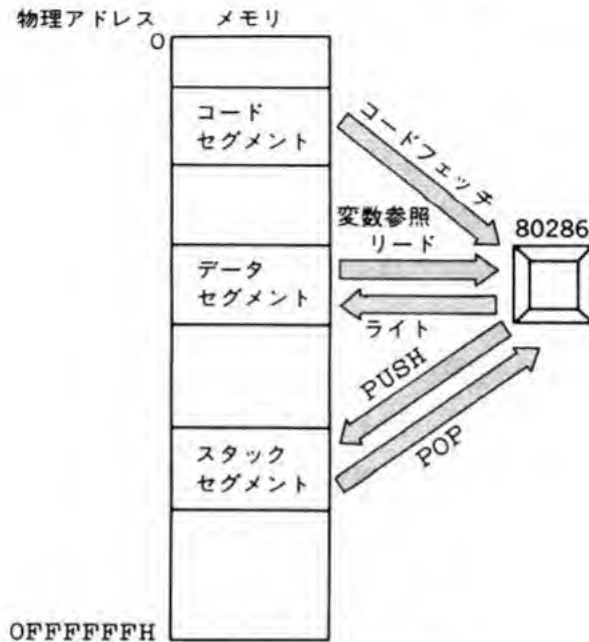


図 1・9 80286 メモリ空間

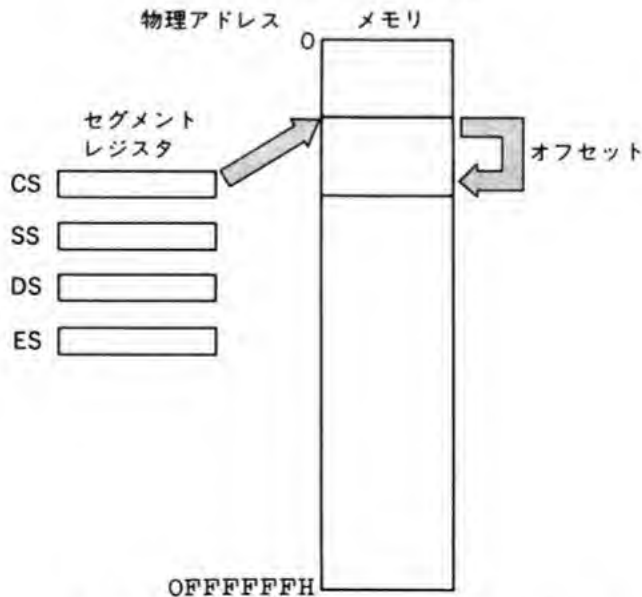


図 1・10 セグメントレジスタとメモリの関係

表 1・2 セグメントレジスタの使用

メモリ参照の形式	使用セグメントレジスタ	オフセット
コードフェッチ	CS	IP
スタックへの PUSH, POP	SS	SP
BP を使用した間接的 データ参照	SS	命令のオペランドで指定する。 ただし、BP を使用して、間接的にオフセットを指定する場合に限る。 また、セグメントオーバーライドプリフィックスを命令の前に付けることによって、SS 以外の DS, ES, CS の任意のセグメントレジスタを使用することが可能である。
データ参照	DS	命令のオペランドで指定する。 ただし、BP を使用した間接的オフセット指定は除く。 また、セグメントオーバーライドプリフィックスを命令の前に付けることによって、DS 以外の ES, SS, CS の任意のセグメントレジスタを使用することが可能である。

BP を間接的に使用してデータ参照を行う場合も、セグメントセクタを指定するために SS が使用される。BP を間接的に使っていないデータ参照は DS がセグメントセクタを指定するために使用される。しかし、データ参照の場合は、このようにセグメントレジスタの使い方が固定されたままでは、使いにくい。したがって、命令の前に、**セグメントオーバーライドインストラクションプリフィックス**と呼ばれる1バイトのコードを追加することによって、暗黙的なセグメントレジスタの使用規則にかかわらず、DS, ES, SS, CS の中から任意のセグメントレジスタを使用することができる。

重要なことは、命令の中でセグメントセクタの値を直接指定するのではなく、セグメントレジスタの種類を指定することである。このことは大きな利益をもたらす。セグメントレジスタに、最初、セグメントセクタの値を代入してお

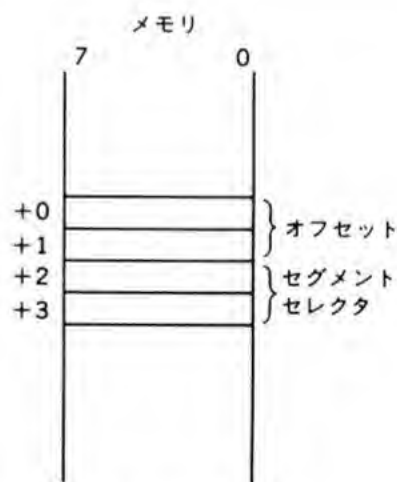


図 1・11 ポインタ

## 1 80286 の 概 要

けば、後はオフセットだけを指定することによって、一度にメモリを参照することができ、たとえば

```
MOV AX , WORD PTR 10
```

はDSで指定されるセグメントのオフセット 10 から始まる1ワードのデータをAXに転送する命令である。AXに転送する値がメモリ内のセグメントセクタ=123、オフセット=10(123:10のように表現する)に定義されている場合は、この命令を含むプログラムを実行する前に、DSに123を初期設定しておく必要がある。

次に、このデータセグメントがセグメントセクタ=234で指定されるセグメントに再配置された場合を考えると、上の命令を含めて、プログラムを書き換える必要はない。DSに123ではなく、234を初期設定するだけでよい。

プログラムにおいて、セグメントセクタとオフセットの両方の値を指定しなければならない場合もある。このとき、図1・11のように、メモリにオフセットとセグメントセクタとを連続に定義して使用できる。この32ビットのデータをポインタと呼ぶ。

## 1-6 リアルモードとプロテクトモード

〔1〕 **80286 動作モード** 80286 は図 1・12 に示すように 2 つの動作モードをもつ。一方を**リアルアドレスモード**と呼び、他方を**プロテクティッドバーチャルアドレスモード**と呼ぶ。リアルアドレスモードという言葉は呼びにくいので、**リアルモード**と呼ぶことにする。リアルモードでは、80286 を高速の 8086 として使用することができる。このため、リアルモードはまた、**86 モード**と呼ばれることもある。ただし、リアルモードでは、80286 のすべての機能を使用することができない。たとえば、アドレスバスは 24 ビットのうち、下位 20 ビットしか使用できない。したがって、リアルモードでのメモリ空間は 8086 と同様の 1M バイトの大きさとなる。

80286 の強力な能力を使用するためにはプロテクティッドバーチャルアドレスモードで使用する。このモードの名前も長く呼びにくいので、**プロテクトモード**と呼ぶことにする。プロテクトモードは、80286 固有の特徴を引き出せるモードであるため、**286 モード**と呼ぶこともある。リアルモード、プロテクトモードの制御は、MSW の PE によって実行する。PE = 0 のとき、80286 はリアルモードで動作し、PE = 1 のとき、プロテクトモードで動作する。

図 1・12 に示したように、80286 をリセットした直後は PE = 0 であり、リアルモードで動作する。リアルモードからプロテクトモードに変更するためには MSW の PE を 1 にする。MSW は図 1・13 に示す LMSW 命令を使用して、データをロードする (MSW へ書き込む) こともできるし、逆に、SMSW 命令を使用して、MSW のデータをストアする (MSW から読み出す) こともできる。MSW の PE だけを 1 に変更するためには、図 1・14 に示すような処理を実行する。ただし、ここに示した処理を実行するだけではプロテクトモードでの正しい実行はできない。プロテクトモードへの完全な初期化処理は 3・7 で述べる。

〔2〕 **リアルモードの論理アドレス** リアルモードとプロテクトモードの違いは表 1・3 のようにまとめることができる。命令体系、機能においては、プロテクトモードはリアルモードの命令体系、機能に新しいものが追加された形になっている。表 1・3 に示すそれぞれの特徴の中で、最も顕著なものはメモリ空間の大きさである。

## 1 80286 の 概 要

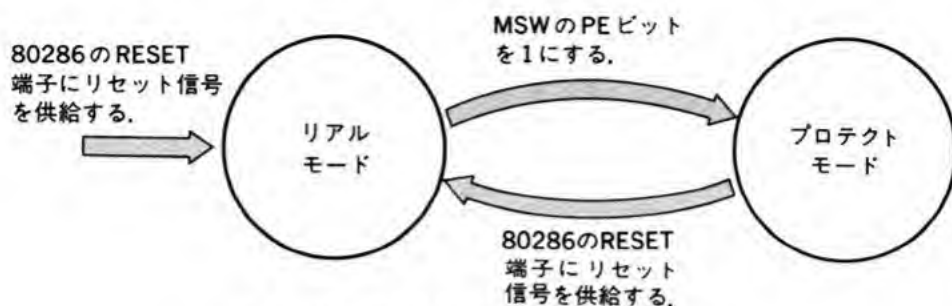


図 1・12 リアルモードとプロテクトモード

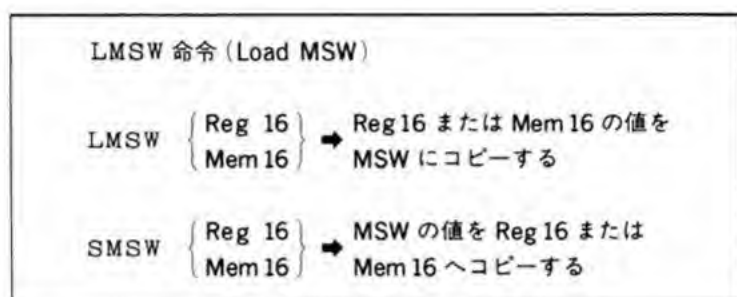


図 1・13 LMSW 命令と SMSW 命令

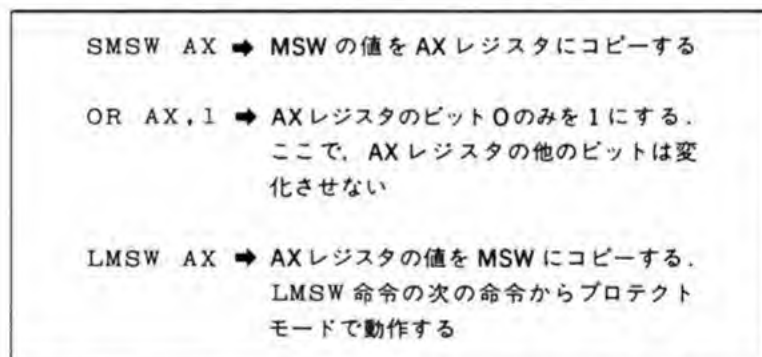


図 1・14 リアルモードからプロテクトモードへの切り換え

80286 内部ではメモリのアドレスを表現するとき、論理アドレスを使用する。80286 は論理アドレスから 24 ビットの物理アドレスを作り、アドレスバスへ出力するのであるが、この方法はリアルモードの場合と、プロテクトモードの場合とではまったく異なる。ここでは、リアルモードにおける、論理アドレスから物理アドレスへの変換について述べる。

セグメントレジスタには、それぞれ、図 1・15 に示すようなセグメントキャッシュが付属する。セグメントキャッシュは 48 ビットのレジスタで、16 ビットのリミットフィールド、24 ビットのベースアドレスフィールド、そして、8 ビット

## 1-6 リアルモードとプロテクトモード

表 1-3 リアルモードとプロテクトモードの基本的特徴

項 目	リアルモードの特徴	プロテクトモードの特徴
データバス	16 ビット	16 ビット
アドレスバス	20 ビット (A <sub>19</sub> -A <sub>0</sub> を使用) メモリ空間 = 1M バイト	24 ビット (A <sub>23</sub> -A <sub>0</sub> を使用) 実メモリ空間 = 16 M バイト
使用レジスタ	AX, BX, CX, DX SI, DI SP, BP, IP FLAG CS, DS, SS, ES MSW, GDTR, IDTR	AX, BX, CX, DX SI, DI SP, BP, IP FLAG CS, DS, SS, ES MSW, GDTR, IDTR LDTR, TR
命令と機能	80186 と共通の命令 + LGDT 命令, SGDT 命令 LIDT 命令, SIDT 命令 LMSW 命令, SMSW 命令 CLTS 命令	80286 リアルモードと共通の命令 + LLDT 命令, SLDT 命令 LTR 命令, STR 命令 LAR 命令, LSL 命令 VERR 命令, VERW 命令 ARPL 命令 特権レベルによるメモリ, I/O 管理 タスクスイッチ命令 仮想メモリのためのサポート

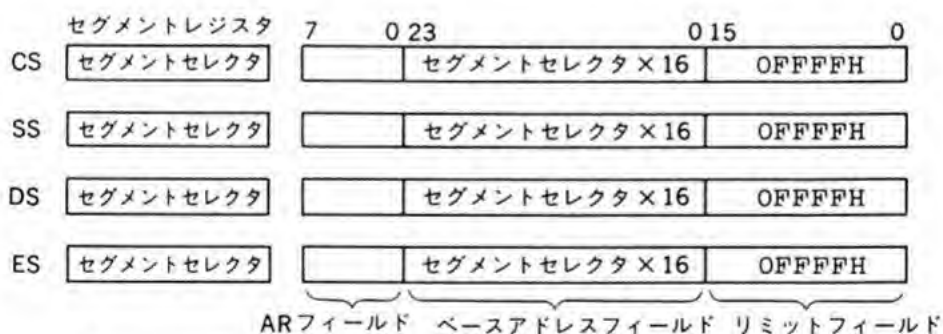


図 1-15 リアルモードにおけるセグメントレジスタとセグメントキャッシュ

の AR (アクセスライト) フィールドに分離される。

リミットフィールドはセグメントの最大のオフセットを定義するが、リアルモードでは、0FFFFH に固定であり、変更することはできない。ベースアドレスフィールドは、セグメントのオフセット 0 の物理アドレスを定義する。80286 はベースアドレスフィールドの値にオフセットを加算した物理アドレスをアドレス

## 1 80286 の 概 要

バスに出力する。しかし、リアルモードではベースアドレスフィールドの値を任意に定めることはできず、必ず、セグメントレジスタの値を 16 倍した値が自動的に設定される。セグメントレジスタが書き換えられたときに、自動的にベースアドレスフィールドも更新される。したがって、リアルモードにおいて、8086 のように論理アドレス 0FFF0H:0400H が論理アドレス 0000H:0300H と同じ物理アドレスを指定するようにするためには、 $A_{20}$ - $A_{23}$  の 4 ビットが常に 0 になるように、外部にハードウェアを付加しなければならない。

AR フィールドにはセグメントの属性を表すコードが入るが、リアルモードにおいては使用されない。

### 8086 の論理アドレス

8086 および 80286 は本質的にリロケートブル（再配置可能）なプログラムができるように作られている。リロケートブルとは、プログラムをメモリの別の位置に移し換えても、書き換えることなく動作することを意味する。

その手段としてセグメンテーションが 8086 から採用された。プログラムでメモリを参照するとき、アドレスをある基準位置からのオフセット（相対アドレス）で表す。メモリの基準位置（ベース）はセグメントレジスタと呼ばれる 4 種類の 16 ビットレジスタに定義し、ベースからオフセットで参照できる領域をセグメントと呼ぶ。したがって、8086 の任意のアドレスはセグメントとオフセットの組み合わせによって表現され、これを論理アドレスと呼ぶ。

論理アドレスから物理アドレスの計算は、セグメントの値を 4 ビットだけ左シフトしてからオフセットを加算する。このため、8086 のセグメントのベースアドレスの下位 4 ビットは必ず 0 である。したがって物理アドレスの 0 番地から 16 バイトごとにセグメントのベースを決めることができる。8086 ではこの 16 バイトの境界をパラグラフと呼び、セグメントレジスタの値をパラグラフ番号とも呼ぶ。

## 2. リアルモードでの使用

リアルモードで使用する 80286 は、第 1 章で述べたように高速の 8086 として使用することができる。8086 のプログラミングに習熟したプログラマは、80286 をリアルモードで使用する場合にも同様にプログラムすることができる。8086 で動いていたプログラムを、そのまま 80286 上で走らせることもできる。ここでは、リアルモードで利用できる 80286 の命令について説明する。この章で述べる命令はプロテクトモードでも使用でき、リアルモード、プロテクトモードの両モードにおいてプログラムを書くための基本的な命令である。

### 2-1 メモリ、I/O 参照と転送命令

[1] **メモリ、I/O のアドレス指定** 80286 で扱うことのできるデータには、図 2・1 に示すように 8 ビットのバイト、16 ビットのワード、32 ビットのダブルワードがある。ダブルワードは主にポインタデータを格納するために使用される。2 バイト以上のデータをメモリに配置するときは、すべて下位バイトを小さいアドレスに、上位バイトを大きいアドレスに連続に配置する。I/O 空間にはインタフェースのレジスタが並んでいるものと考えればよい。I/O との間では、図 2・2 に示すようにバイト、ワードデータを入出力することができる。ワードデータは、やはり下位バイトが I/O の小さいアドレスに、上位バイトが I/O の大きいアドレスになるよう配置する。

メモリのオフセットを指定する方法は、図 2・3 に示すような方法がある。図 2・3(1) に示す方法は**直接指定**と呼び、直接に定数でオフセットを指定する。アセンブリ言語の命令では、データのタイプも表現する必要があるので

BYTE PTR 10 ➡ オフセットが 10 のメモリに配置した  
バイトデータ

WORD PTR 10 ➡ オフセットが 10 のメモリに配置した  
ワードデータ

DWORD PTR 10 ➡ オフセットが 10 のメモリに配置した  
ダブルワードデータ

のように書く。ここで、大きさが 2 バイト以上のデータのオフセットは、そのもっとも下位のバイトのオフセットで表す。図 2・3(2) に示す**間接指定**は、**ベースレジスタ**または**インデックスレジスタ**を用いて、間接的にオフセットを指定する方法である。ベースレジスタとは、BX、BP の総称であり配列などの構造的データの基底アドレスを表すのに用いられる。インデックスレジスタは、SI、DI の総称であり、配列、ストラクチャなどの構造的データの各要素を間接的に表すために用いられる。間接指定はベースレジスタ、インデックスレジスタ、ディスプレイスメント（定数で表した変位）の 3 要素の任意の組み合わせによって、さまざまな構造のデータを柔軟に指定することができる。間接指定の 3 要素のうち、指定された要素の和がオフセットになる。

## 2-1 メモリ、I/O 参照と転送命令

I/O にはセグメンテーションはなく、0 から 65535 までの I/O アドレスを指定する。I/O アドレスの表現も、図 2・4 に示すように直接指定と間接指定があるが、直接に指定できるのは、0 から 255 までである。

〔2〕 **転送命令** レジスタ、メモリ、I/O のデータ転送は、図 2・5 に示すような命令で行われる。メモリ-レジスタ間、レジスタ-レジスタ間のデータ転送は MOV 命令または XCHG 命令を使用する。MOV 命令、XCHG 命令の書き方を図 2・6 に示す。MOV 命令は 2 つのオペランドをもち、オペランド 2 の値をオペ

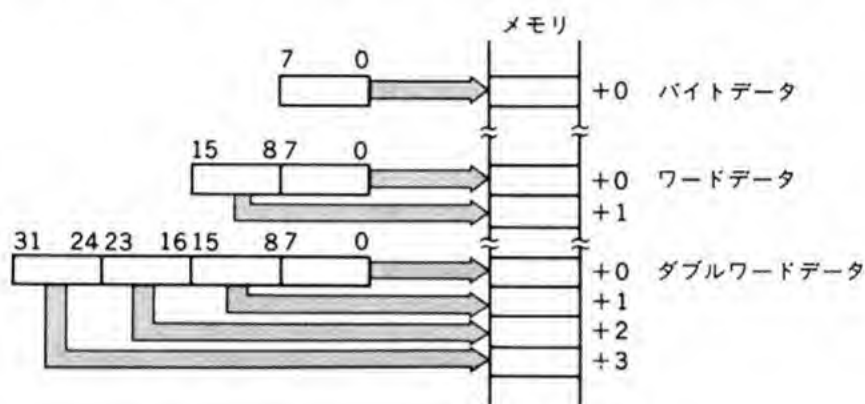


図 2・1 80286 のデータとメモリへの配置

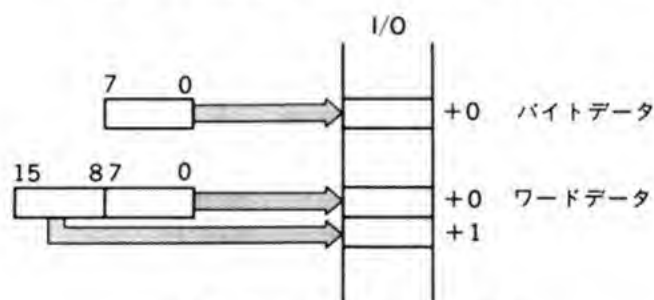


図 2・2 I/O へのデータの配置

(1) **直接指定**

オフセット = 定数

(2) **間接指定**

オフセット = [ベースレジスタ] + [インデックスレジスタ] ± ディスプレイスメント

ベースレジスタ = BX または BP

インデックスレジスタ = SI または DI

ディスプレイスメント = ベースレジスタ + インデックスレジスタからの相対変位を表す符号付き定数

図 2・3 アドレッシングモード

## 2 リアルモードでの使用

ランド1に代入する。オペランド1にDS, ES, SSのセグメントレジスタを指定することもできるが、このときオペランド2に定数を指定することはできない。左側のオペランドが値を代入する先(デスティネーション)を表す。インテルのアセンブリ言語の命令では、MOV命令に限らず2つのオペランドをもち、どちらかが値を更新される場合、値を更新するオペランドを必ず左側に書くようにする。ここで、オペランドは、バイトタイプでもワードタイプでもよいが、両方のオペランドのタイプは一致していなければならない。また、両方のオペランドに

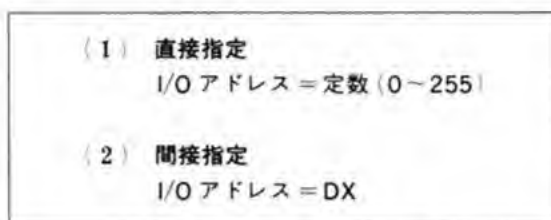
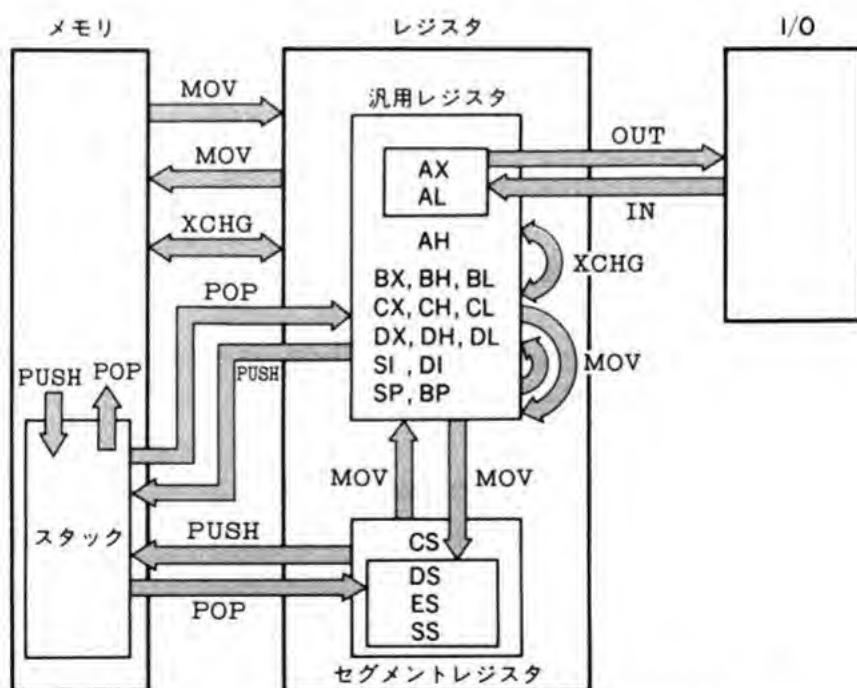


図 2・4 I/O のアドレス指定



ただし、転送するデータのタイプはバイトまたはワード (= 2 バイト) であり、転送元と転送先のデータタイプは一致しなければならない。

また、スタックへの **PUSH**, **POP** で転送されるデータタイプは常にワードである。

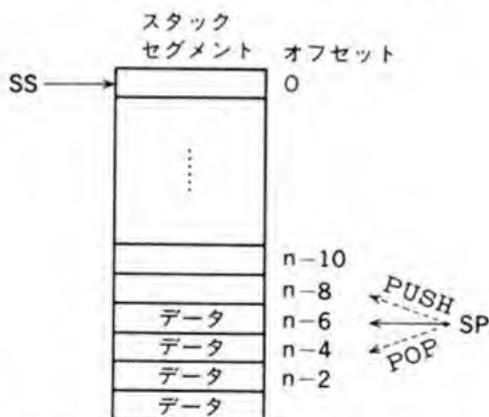
図 2・5 データ転送

同時にメモリを指定することは許されない。XCHG 命令は、2つのオペランドの値を相互に入れ換える命令である。

〔3〕 **スタックの使用** 図2・5において、スタックはデータを保存するためのセグメントであるが、図2・7に示すように、PUSH, POP と呼ばれる方法で



図 2・6 MOV, XCHG 命令



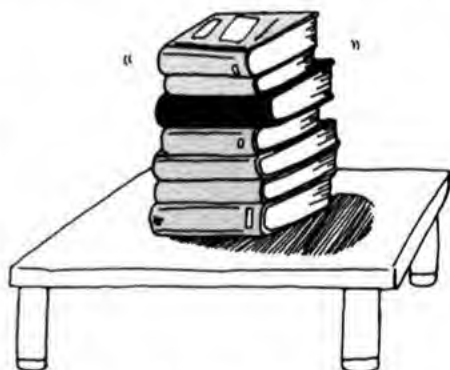
- (1) **PUSH** オペランド  
 オペランド = ワードメモリ, ワード汎用レジスタ  
 CS, DS, ES, SS, 定数  
 PUSH の操作:  $SP \leftarrow SP - 2$   
 $WORD PTR SS:[SP] \leftarrow \text{オペランド}$
- (2) **POP** オペランド  
 オペランド = ワードメモリ, ワード汎用レジスタ  
 DS, ES, SS  
 POP の操作:  $\text{オペランド} \leftarrow WORD PTR SS:[SP]$   
 $SP \leftarrow SP + 2$

図 2・7 スタックと PUSH, POP 操作

## 2 リアルモードでの使用

データを出し入れする。スタックは、机の上に本を積むようにデータを保存する。このようにすると、スタックのトップに最も新しいデータを見ることができる。スタックセグメントはオフセットの高い領域から低い領域へと使用する。スタックトップのオフセットはSPによって指定する。PUSHはスタックに新しいデータを積む操作である。このとき、SPは自動的に2だけ減算される。POPは逆にスタックトップのデータを取り出す操作である。この場合、SPは2だけ加算される。PUSH、POPの操作は、PUSH命令、POP命令を使用してデータを保存するときに実行される。また、CALL命令、割り込み、RET命令、IRET命令で、主プログラムへの戻りアドレスを保存するために自動的にPUSH、POPが実行される。PUSH、POPで扱われるデータは必ずワードタイプでなければならない。

〔4〕 I/O へのデータ転送 I/Oとのデータの入出力は、図2・8に示すようにIN命令、OUT命令を使用する。I/Oアドレスは直接に定数で指定してもよいし、DXで間接的に指定することもできる。ただし、定数でI/Oアドレスを指定するときは0から255までのアドレスしか指定できない。IN命令、OUT命令では、バイト、ワードの2種類のデータを扱うことができるが、このタイプの区別は、命令のオペランドにALを使用するかAXを使用するかによって区別できる。



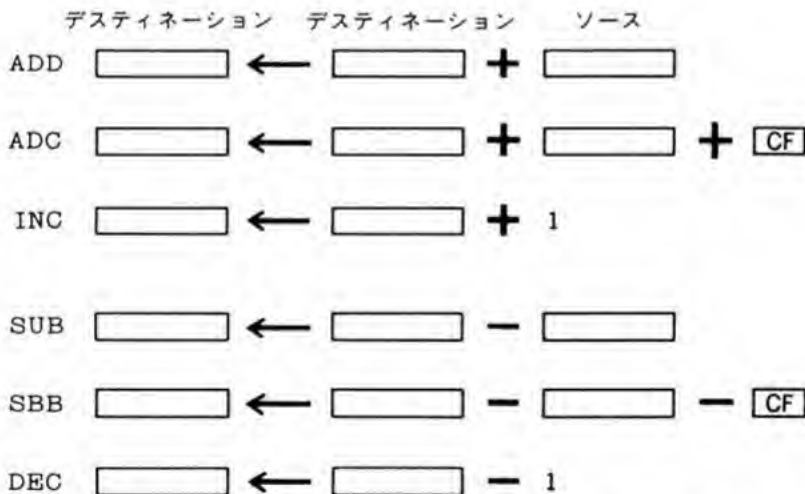
本のスタック

- |                            |
|----------------------------|
| (1) IN 命令                  |
| IN { AL<br>AX }, I/O アドレス  |
| (2) OUT 命令                 |
| OUT I/O アドレス, { AL<br>AX } |

図 2・8 IN 命令と OUT 命令

## 2-2 演 算 命 令

演算命令には算術演算、論理演算、シフト、ローテイトがある。算術演算の中で、加算、減算は図 2・9 に示すものがある。ADC 命令、SBB 命令はキャリーフラグ (CF) を含めた演算で、32 ビットのデータの加算、減算などに使用する。たとえば、オフセットが 0 のメモリにある 32 ビットデータと、オフセットが 4 のメモリにある 32 ビットデータを加算し、結果をオフセットが 8 のメモリに代入する処理は、図 2・10 に示すようになる。32 ビットデータの演算では、下位ワード、上位ワードを別々に処理するが、上位ワードの演算において下位ワードからの桁上げも含めて演算しなければならない。したがって、上位ワードの加算は、下位ワードの加算の後に実行し ADC 命令を使用する。このことは、減算の場合も同様である。減算のときは、下位ワードの減算で引きすぎが発生した場合、CF が 1 となるから、上位ワードの減算には SBB を使用すればよい。



命令の形式：命令 デスティネーションオペランド，ソースオペランド

ここで、デスティネーションはバイトまたはワードタイプの汎用レジスタまたはメモリ。

ソースはバイトまたはワードタイプの汎用レジスタまたはメモリ。定数を指定することもできる。

ただし、デスティネーション、ソースのタイプは一致しなければならない。

また、メモリ-メモリ間の演算はできない。

図 2・9 加算と減算

## 2 リアルモードでの使用

乗算と除算の命令を図2・11に示す。MUL 命令、DIV 命令では、データは符号なしとして扱われる。IMUL 命令、IDIV 命令では、データの最上位ビットが符号ビットで、マイナスデータは2の補数表現で扱われる。また、IMUL にはワード汎用レジスタ、ワードメモリに置かれた符号付き整数を定数で乗算できるものがあり、この命令は3つのオペランドをもつ。ただし、左側の2つのオペランドが同じレジスタの場合は、右側の2つのオペランドだけを指定する。

ローテイト、シフト演算には、図2・12に示す種類がある。命令の書き方はす

```
MOV AX , WORD PTR 0
MOV DX , WORD PTR 2
ADD AX , WORD PTR 4
ADC DX , WORD PTR 6
MOV WORD PTR 8 , AX
MOV WORD PTR 10, DX
```

図 2・10 32ビットデータの加算

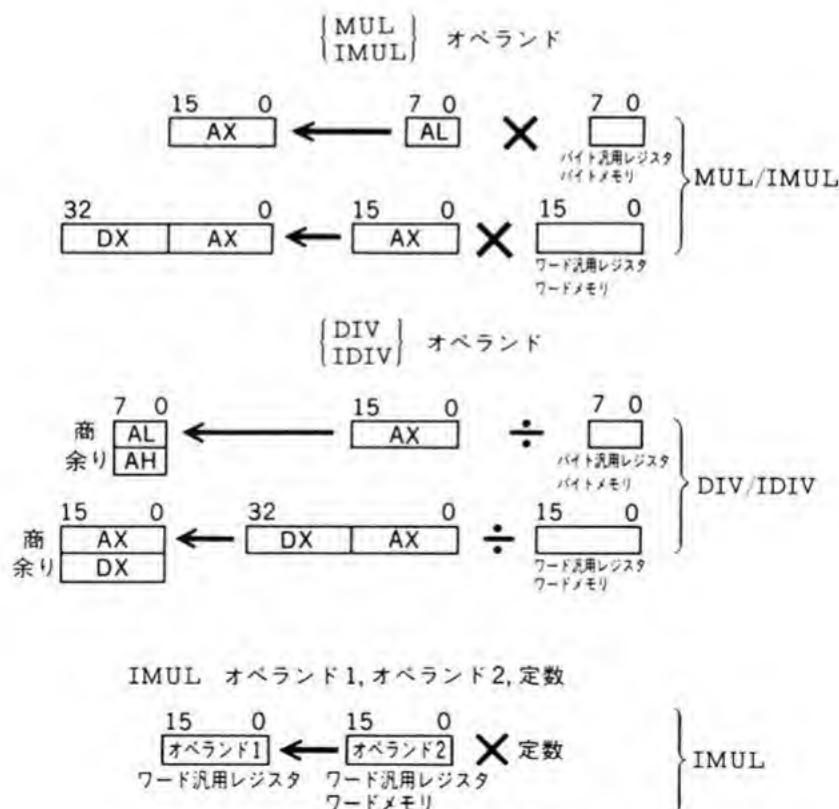
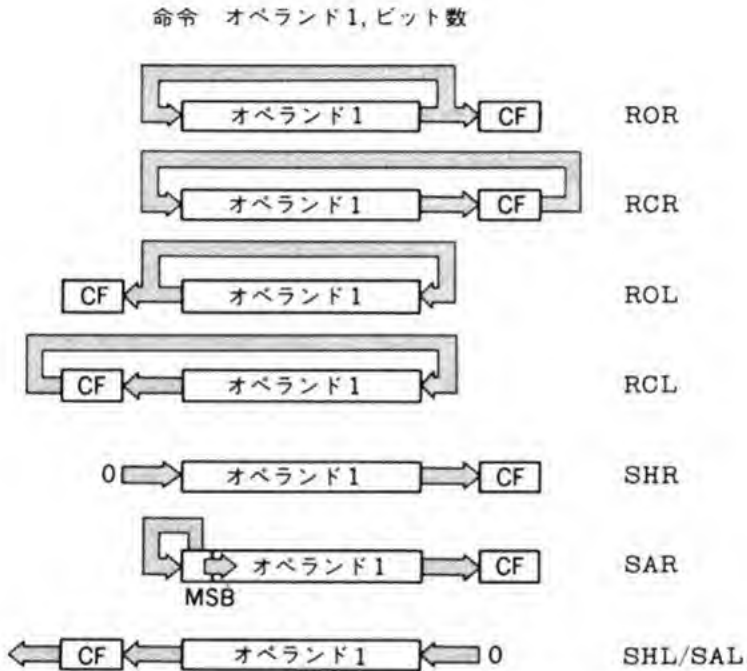


図 2・11 乗算と除算

## 2-2 演 算 命 令

べて同じである。図 2・12 に示すように、オペランド 1 にはバイト、またはワードタイプの汎用レジスタ、あるいはメモリを指定し、右のオペランドにローテイト、またはシフトするビット数を指定する。ビット数は定数で直接指定してもよいし、CL で間接的に指定することもできる。



オペランド 1 には、バイトまたはワードタイプの汎用レジスタ、メモリを指定できる。

図 2・12 ローテイト、シフト演算

### 2-3 制御命令

プログラムの処理の流れを変える命令には、図 2・13 に示すように JMP 命令、CALL 命令、RET 命令がある。これらの命令はその機能によって、さらに何種類かに分類される。

〔1〕 **JMP 命令** JMP 命令は、そのジャンプできる距離によって、3 種類に分類できる。それを、識別するために、**short**, **near**, **far** の言葉を使用する。short JMP は、JMP 命令の次の命令のアドレスを中心に +127～-128 バイトの範囲にジャンプすることができる。near JMP は、JMP 命令の次の命令のアドレスを中心に  $(2^{15}-1) \sim -2^{15}$  バイトの範囲にジャンプすることができる。オフセットの計算ではビット 15 からのキャリーは無視されるから、near JMP を使用すればセグメント内の任意のアドレスへジャンプすることができる。far JMP は、命令の中にジャンプする先のセグメントセレクトとオフセットを指定し、80286 のメモリ空間の任意のアドレスにジャンプすることができる。short JMP, near JMP が IP しか書き換ええないのに対して、far JMP は CS と IP を書き換えてジャンプする。

〔2〕 **CALL 命令と RET 命令** CALL 命令にも JMP 命令と同様に near と far の区別がある。CALL 命令は、指定されたアドレスにジャンプする前に CALL 命令の次の命令のアドレスを戻りアドレスとしてスタックへ PUSH する。near CALL では、near JMP と同様に CS の値は変化しないから、図 2・14(a) に示すように戻りアドレスとしてオフセットだけを PUSH する。far CALL の場合は CS と IP の値がともに変化するから、図 2・14(b) に示すようにセグメントセレクト、オフセットの順で 2 ワードの戻りアドレスを PUSH する。

CALL 命令で制御を移した手続きの中で RET 命令を実行すると、CALL 命令の次の命令へ戻ることができる。CALL 命令に near と far の 2 種類があるのに対して、RET 命令にも near, far の 2 種類がある。near RET はスタックトップから 1 ワードを POP して、IP に代入することによって制御を変える。far RET は、スタックトップから 2 ワードを POP して IP と CS に代入して制御を変える。したがって、near CALL に対して near RET, far CALL に対して far RET のように対にして使用しなければ正しい制御の変更はできない。ここでは、JMP

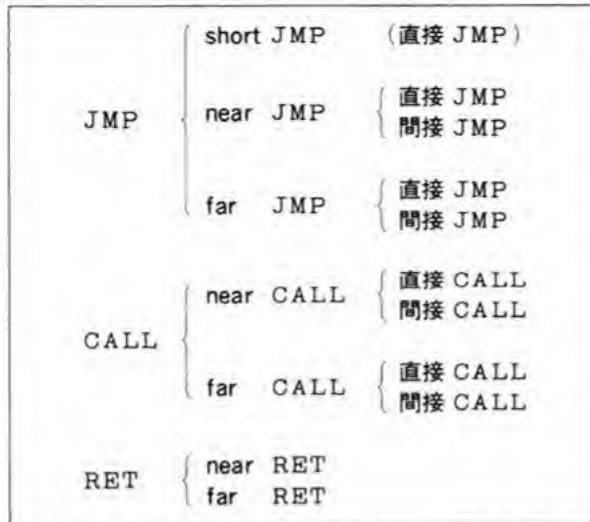


図 2・13 JMP 命令, CALL 命令, RET 命令の種類

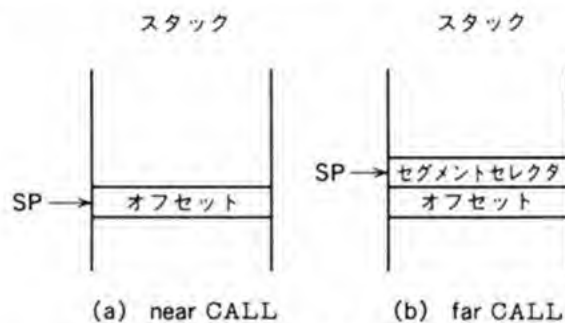


図 2・14 CALL 命令実行後のスタック

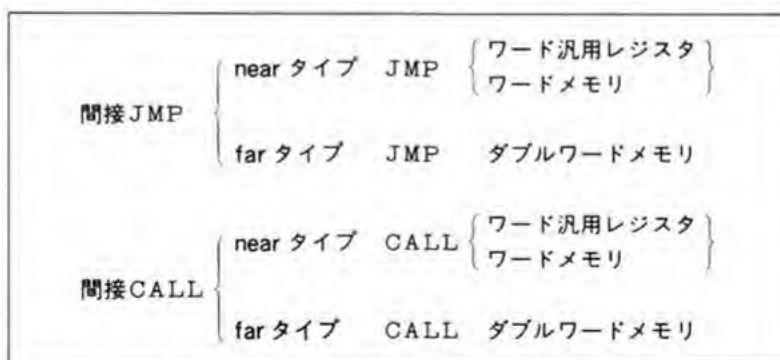


図 2・15 間接 JMP 命令と間接 CALL 命令

命令, CALL 命令, RET 命令のタイプを明確に表現するために, 命令の前に short, near, far と記したが, アセンブリ言語 ASM 286 では short, near, far を書く必要はない。

## 2 リアルモードでの使用

〔3〕 間接的な JMP, CALL 命令      JMP 命令, CALL 命令にはそれぞれ、間接 JMP, 間接 CALL と呼ばれるものがある。この命令の書き方を図 2・15 に示す。これらの命令は、オペランドに指定したワード汎用レジスタまたは変数によって、間接的に制御を移すアドレスを指定する。間接 JMP, 間接 CALL にも near タイプのものと far タイプのものがある。オペランドにワード汎用レジスタまたはワードメモリを指定した命令は near タイプであり、そのレジスタまたはメモリの値を IP に代入して制御を移す。オペランドにダブルワードメモリを指定したものは far タイプであり、ダブルワードメモリの上位ワードを CS に代入し、下位ワードを IP に代入して制御を移す。

〔4〕 条件 JMP 命令      アセンブリ言語のプログラムで分岐処理を定義する命令が条件 JMP である。表 2・1 に単一フラグによる条件 JMP 命令を示す。状態フラグの値によってオペランドに指定したラベルへジャンプする。フラグの値が条件と逆の場合は次の命令を実行する。プログラムでは 2 つの値の大小比較の結果によって、分岐する処理を書くことが多い。そのような処理を簡単に表現できるように 2 つ以上のフラグの値による条件 JMP が用意されている。これらを表 2・2 に示す。これらの条件 JMP は通常、CMP 命令によって 2 つの値を比較した後に書く。符号なし整数の比較の場合は、"Above (より大きい)"、"Below (より小さい)"、"Equal (等しい)"を表す、"A"、"B"、"E"、それとその否定を表す"N"を"J"の後に組み合わせて命令のニーモニックを作る。また、符号付き整数の比較の場合は、"Greater (より大きい)"、"Less (より小さい)"、"Equal (等しい)"を表す、"G"、"L"、"E"と否定を表す"N"を"J"の後に組み合わせて命令のニーモニックを作る。

プログラムでは、ある処理を何回か繰り返し実行することがよくある。このような繰り返し処理は上に述べた CMP 命令と条件 JMP を使用してももちろん記述できるが、図 2・16 に示すように LOOP 命令を使用してもよい。LOOP 命令は CX から 1 を減算する処理を実行し、ZF が 1 であればオペランドに示したラベルへジャンプする。したがって、図 2・16 のようにラベル LABEL 1 から LOOP 命令までの処理を CX に初期設定した回数だけ繰り返し実行する。LOOP 命令で注意すべきことは、ZF を調べる前に CX から 1 を減算することである。したがって、CX の初期値が 0 の場合は、繰り返し回数は 0 回ではなく、65536 回になる。CX の値が 0 であるのに対して繰り返し回数を 0 回にしたければ、JCXZ

表 2・1 単一フラグによる条件 JMP

フラグビット	条件 JMP 命令	命令の形式	ジャンプする条件
CF	JC	JXXX ラベル	CF = 1
	JNC		CF = 0
PF	JP/JPE		PF = 1
	JNP/JPO		PF = 0
ZF	JZ/JE		ZF = 1
	JNZ/JNE		ZF = 0
SF	JS		SF = 1
	JNS		SF = 0
OF	JO		OF = 1
	JNO		OF = 0

表 2・2 複数フラグによる条件 JMP

CMP X,Y  
条件 JMP 命令 ラベル

	条件 JMP 命令	ジャンプする条件
符号なし 整数の 比較	JA/JNBE	X > Y
	JAE/JNB	X ≥ Y
	JE/JZ	X = Y
	JBE/JNA	X ≤ Y
	JB/JNAE	X < Y
符号付き 整数の 比較	JG/JNLE	X > Y
	JGE/JNL	X ≥ Y
	JE/JZ	X = Y
	JLE/JNG	X ≤ Y
	JL/JNGE	X < Y

JCXZ NEXT
LABEL1:
...
...
LOOP LABEL1 → DEC CX
NEXT:                    等価    JZ LABEL1

図 2・16 LOOP 命令と JCXZ 命令

## 2 リアルモードでの使用

命令を使用して繰り返し処理をスキップするように書くことができる。ここで、JCXZ 命令は、CX が 0 のときオペランドに指定したラベルへジャンプする条件 JMP である。

また、以上述べてきたすべての条件 JMP と LOOP 命令は、short JMP と同様に次の命令のアドレスを中心にして、-128 ~ +127 バイトの範囲にしかジャンプできない。もし、条件によって short JMP 以上の距離に制御を移行したい場合は、JMP 命令と組み合わせて 2 回ジャンプを作るようにすればよい。

### ジャンプ命令

ジャンプ命令のコードは、2 バイト、3 バイト、5 バイトの 3 種類があり、それぞれショートジャンプ、ニアジャンプ、ファージャンプ（ロングジャンプと書いている本もある）と呼ぶ。

本文で述べたように、近くに制御を移行させる場合にはジャンプ命令のコードを短くできるが、ユーザがジャンプ命令を書くアドレスと制御の移行先の距離を予想して、採用するジャンプ命令の種類を決めることはプログラミングにおいて適当ではない。インテルのアセンブラ ASM 86 および ASM 286 では

JMP    制御移行先のラベル

のように書くだけでよい。ジャンプ命令の種類はアセンブラが自動的に最適なものを選択してくれる。しかし、ラベルが前方参照の場合のようにアセンブラが自動的にその種類を決定できないとき

JMP SHORT    ラベル

JMP NEAR PTR    ラベル

JMP FAR PTR    ラベル

のようにユーザが明示することもできる。

## 2-4 ストリング命令

〔1〕 ストリング命令の動作と種類 プログラムにおいて、メモリに連続して定義されるデータを扱うことが多い。たとえば、メモリの 8000H 番地から始まる 100H バイトのデータを、メモリの 9000H 番地から 90FFH 番地までの領域に転送するいわゆるブロック転送がある。80286 では、このようなメモリブロックを扱う上で有効なストリング命令と呼ばれるいくつかの命令が用意されている。表 2-3 にストリング命令の一覧を示す。ストリング命令のオペコードニーモニックは最後に S と、バイト単位の転送かワード単位の転送かによって

表 2-3 ストリング命令

オペコード ニーモニック	動 作
MOVSB MOVSW	DS:SI で指定されるメモリの 1 バイトまたは 1 ワードの値を ES:DI で指定されるメモリに転送してから SI, DI を 1 または 2 だけ増加または減少させる。
CMPBSB CMPBSW	DS:SI で指定されるメモリの 1 バイトまたは 1 ワードの値から ES:DI で指定されるメモリの 1 バイトまたは 1 ワードの値を減算してから、SI, DI を 1 または 2 だけ増加または減少させる。このときメモリの値は変化せず FLAG の値が変化するだけである。
SCASB SCASW	AL または AX から ES:DI で指定されるメモリの 1 バイトまたは 1 ワードの値を減算してから、DI を 1 または 2 だけ増加または減少させる。このとき AL, AX の値は変化せず、FLAG の値が変化するだけである。
LODSB LODSW	DS:SI で指定されるメモリの 1 バイトまたは 1 ワードの値を AL, AX に代入してから、SI を 1 または 2 だけ増加または減少させる。
STOSB STOSW	AL または AX の値を ES:DI で指定されるメモリの 1 バイトまたは 1 ワードに代入してから、DI を 1 または 2 だけ増加または減少させる。
INSB INSW	DX で指定される I/O アドレスの 1 バイトまたは 1 ワードの値を ES:DI で指定されるメモリに転送してから、DI を 1 または 2 だけ増加または減少させる。
OUTSB OUTSW	DS:SI で指定されるメモリの 1 バイトまたはワードの値を DX で指定される I/O に出力してから、SI を 1 または 2 だけ増加または減少させる。

ここで、オペコードニーモニックが B で終わっているときはバイト転送であり、アキュムレータとして AL を使用し、インデックスレジスタの増加、減少分は 1 である。また、オペコードニーモニックが W で終わっているときはワード転送であり、アキュムレータとして AX を使用し、インデックスレジスタの増加、減少分は 2 である。

FLAG の DF が 0 のときインデックスレジスタを増加させる。FLAG の DF が 1 のときインデックスレジスタを減少させる。DS はセグメントオーバーライドプリフィックスによって、任意のセグメントレジスタに変更することができる。

## 2 リアルモードでの使用

B または W が付く。処理の対象は命令によって暗黙的に決まっているので、オペランドを指定する必要はない。

MOVS 命令は、図 2・17 に示すようにメモリの DS:SI で指定される 1 バイトまたは 1 ワードの値を、ES:DI で指定されるメモリに転送する。MOVSB 命令のときは 1 バイトの転送であり、MOVSW 命令のときは 1 ワードの転送となる。また、このとき SI, DI のインデックスレジスタを自動的に更新する。MOVSB 命令のときは、インデックスレジスタを 1 だけ更新し、MOVSW 命令のときは、インデックスレジスタを 2 だけ更新する。さらに、インデックスレジスタの更新は、FLAG の DF によって増加か減少かが制御される。インデックスレジスタは、DF=0 のとき増加され、DF=1 のとき減少される。

表 2・3 に示す MOVS 命令以外のストリング命令においても同様の動作を実行する。ストリング命令のオペランドは、AL または AX、メモリのバイトまたはワードのデータ、そしてアドレスが DX で指定される I/O であり、ストリング

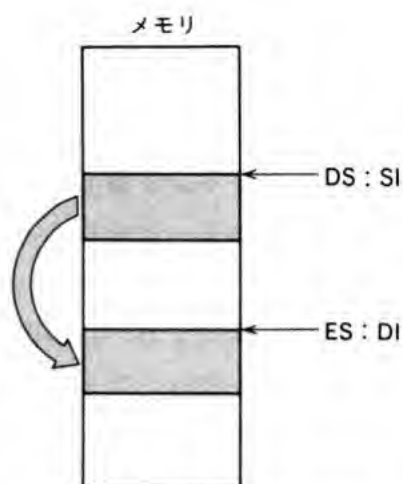


図 2・17 MOVS 命令によるデータ転送

表 2・4 DF の設定

オペコード ニーモニック	動 作
CLD	DF を 0 にクリアする。
STD	DF を 1 にセットする。

```

SOURCE_PTR DD 00008000H
DESTINATION_PTR DD 00009000H
      ⋮
LDS SI, SOURCE_PTR
  ➡ 変数 SOURCE_PTR の上位ワードを DS に代
    入し、下位ワードを SI に代入する
LES DI, DESTINATION_PTR
  ➡ 変数 DESTINATION_PTR の上位ワードを
    ES に代入し、下位ワードを DI に代入する
CLD ➡ DF を 0 にクリアする
MOVSW

```

図 2・18 MOVSW 命令を使用した 1 要素のデータ転送

## 2-4 ス ト リ ン グ 命 令

表 2・5 REP プリフィックスの種類

REP プリフィックスの使用	動 作
REP <span style="border: 1px solid black; padding: 2px;">ストリング命令</span>	REP の直後に指定されたストリング命令を、CX で定義された回数だけ繰り返し実行する。
REPE <span style="border: 1px solid black; padding: 2px;">ストリング命令</span> (REPZ)	REPE の直後に指定されたストリング命令を、CX で定義された回数だけ繰り返し実行する。ただし、ZF = 0 の条件で、CX の値に関係なく、繰り返しを終了する。
REPNE <span style="border: 1px solid black; padding: 2px;">ストリング命令</span> (REPNZ)	REPNE の直後に指定されたストリング命令を、CX で定義された回数だけ繰り返し実行する。ただし、ZF = 1 の条件で、CX の値に関係なく、繰り返しを終了する。

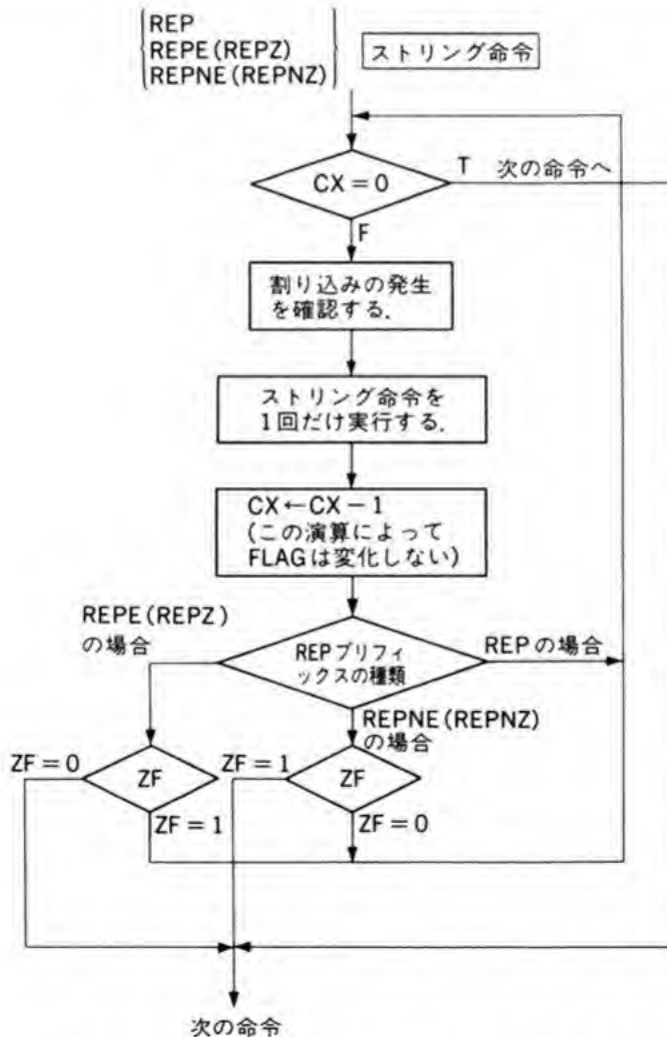


図 2・19 REP プリフィックス付きのストリング命令

## 2 リアルモードでの使用

命令の種類によって2つの処理対象が暗黙的に定義されている。ここで、ストリング命令の処理対象がメモリのとき、データが代入されるメモリは必ず ES:DI によって指定される。逆にソースデータをもつメモリは DS:SI によって指定される。ソースメモリを指定するセグメントレジスタは、一般に DS であるが、セグメントオーバーライドプリフィックスを指定することによって、他の任意のセグメントレジスタに変更することができる。ES を変更することはできない。インデックスレジスタの SI (source index), DI (destination index) の名前はストリング命令における上のような使い方から与えられたものである。ストリング命令以外の命令においては、インデックスレジスタとして SI, DI のどちらを使用してかまわない。

また、ストリング命令に関係するインデックスレジスタは自動的に更新される。MOVS 命令、CMPS 命令においては SI, DI の両方が、また SCAS 命令、STOS 命令、INS 命令においては DI が、LODS 命令、OUTS 命令においては SI がそれぞれ、DF の制御によって増加または減少される。DF=0 のとき、バイト操作のストリング命令では1だけ、またワード操作のストリング命令では2だけ増加される。DF=1 のときは、同様にして減少される。

ストリング命令は、以上のように暗黙的に使用するいくつかのレジスタまたは制御フラグをもつので、ストリング命令を実行する前にそれらを初期設定する必要がある。ここで、FLAG の DF の1ビットだけの初期設定のために、表2・4に示すような CLD 命令、STD 命令が用意されている。たとえば、メモリの 8000H 番地にある1ワードのデータを、メモリの 9000H 番地に転送するためには MOVSW 命令を用いて、図2・18に示すようにプログラムを書く。

〔2〕 **REP プリフィックス付きストリング命令**      しかし、図2・18に示すようにストリング命令だけではデータブロックの1要素しか扱うことができない。表2・5に示す REP プリフィックスをストリング命令の前に書くことによって、ストリング命令を図2・19に示すように繰り返し実行することができる。ここで、実行の繰り返し回数は CX によって定義される。また、REPE (REPZ と書いてもよい) プリフィックスは CX が0にならなくても、ZF が0になればストリング命令の繰り返しを終了する。逆に REPNE (REPNZ と書いてもよい) の場合は ZF が1になれば、CX≠0の状態であっても繰り返しを終了する。REPE, REPNE は、CMPS 命令、SCAS 命令とともに用いる。したがって、メモリの

8000H 番地から定義された 100H ワードのデータを、9000H 番地から始まるメモリにワード単位で転送する処理は図 2・20 に示すようになる。なお、ストリング命令の繰り返し処理には REP プリフィックスを付ける代わりに LOOP 命令を使用してもかまわない。

```

SOURCE_PTR DD 00008000H
DESTINATION_PTR DD 00009000H
          ⋮
LDS SI,SOURCE_PTR
LES DI,DESTINATION_PTR
MOV CX,100H
CLD
REP MOVSW

```

図 2・20 MOVSW 命令を使用した 100H ワードのデータ転送

### セグメントオーバライド(インストラクション)プリフィックス

8086 が論理アドレスから物理アドレスを計算するとき、セグメントレジスタの値をベースアドレスとして使用するが、CS, DS, SS, ES のどのレジスタを使用するかは暗黙的に決まっている。しかし、命令の前に

2EH ➡ 続く命令が CS を使用してメモリ参照をする。

3EH ➡ 続く命令が DS を使用してメモリ参照をする。

36H ➡ 続く命令が SS を使用してメモリ参照をする。

26H ➡ 続く命令が ES を使用してメモリ参照をする。

の 4 種類の 1 バイトのセグメントオーバライドプリフィックスを指定することによって、使用するセグメントレジスタをユーザが任意に明示することができる。

プログラムでセグメントオーバライドプリフィックスを表現するとき

```
MOV AX,ES:WORD PTR 100H
```

のように、メモリオペランドの前に CS:, DS:, SS:, ES: のように書く。しかし、セグメントレジスタの種類を考えながらプログラムを書くことは煩わしいので、ASSUME 宣言文という疑似命令が用意されている。ASSUME 宣言を書くことによってセグメントオーバライドプリフィックスの使用を統一的にアセンブラにまかせることができる。

### 2-5 拡張命令

C 言語の関数、PASCAL 言語および PL/M 言語などの手続きのような、高級言語の手続きについて考えるとき、手続き内で使用する変数をデータセグメントのようなスタティック（静的）な領域に定義する場合と、手続きが引用されている間だけスタックセグメントにダイナミック（動的）に定義する場合がある。静的な変数はプログラムが実行されている間、常にメモリ上に存在するが、動的な変数は部分的な手続きを実行する間だけ、スタックセグメントに一時的に存在するだけである。C 言語の場合、関数で定義するスタティック変数は前者の例であり、オート変数が後者の例である。

80286 は、スタックセグメントに動的な変数領域を割り付ける ENTER 命令と、ENTER 命令によって作成した動的な変数領域をスタックから削除する LEAVE 命令をもつ。図 2-21 に ENTER 命令とその動作を示す。ENTER 命令は 2 つの定数オペランドをもち、一般に手続き定義の先頭を書く命令である。左側のオペランドによって動的な変数領域のバイト数を指定する。右側のオペランドは手続きのネストのレベルを表す。C 言語の関数とか、図 2-22 に示す手続き A のようなもっとも外側の手続きで ENTER 命令を書くときは、レベルを 1 に設定すればよい。

たとえば、スタックセグメントに 100 バイトの動的な変数領域を作るとき

```
ENTER 100,1
```

の命令を手続きの先頭を書けばよい。この命令を実行したとき、図 2-23 に示すような領域がスタックセグメントに定義される。ENTER 命令がスタックセグメントに定義する領域をスタックフレームと呼ぶ。スタックフレームは動的な変数領域とディスプレイ領域からなるが、レベルを 1 に指定した場合は ENTER 命令実行前の BP の値とスタックフレームのベースを示す BP の値 BP<sub>A</sub> が保存されている。また、スタックフレームのベースアドレスは、BP によって指定されるからそれぞれの動的な変数は

```
WORD PTR [BP-4]
```

```
BYTE PTR [BP-6]
```

のように参照することができる。

ここで再び図 2-22 において、手続き B について考える。手続き B において

```
ENTER DYNAMIC, LEVEL
```

### III 等価

```

LEVEL ← LEVEL MOD 32
PUSH BP
TEMP ← SP
if LEVEL > 0

then repeat (LEVEL-1)
    BP ← BP-2
    PUSH WORD PTR [BP]
endrepeat
PUSH TEMP

end if

BP ← TEMP
SP ← SP-DYNAMIC

```

ここで、repeat は repeat(n) の次から endrepeat までの処理を n 回繰り返すものとする。また、TEMP は内部作業のための変数である。

LEVEL MOD 32 は LEVEL を 32 で割った余りを求めることを意味する。

DYNAMIC は、動的変数領域のバイト数を表す 1 ワードの定数であり、LEVEL は手続きのネストを表す 1 バイトの定数である。

図 2・21 ENTER 命令の動作

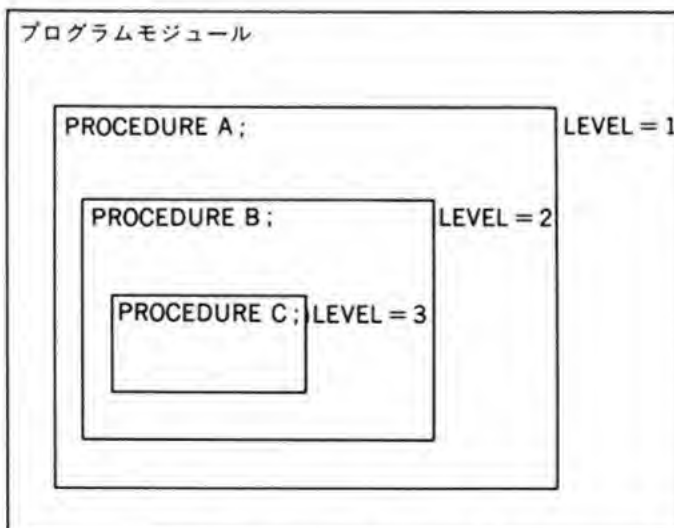


図 2・22 ネスト構造の手続き定義

## 2 リアルモードでの使用

は、手続き A で定義した変数も参照できなくてはならない。そのために、手続き B のスタックフレームに、手続き A のスタックフレームの BP の値  $BP_A$  を記録しておくことが有効になる。したがって、手続き B で 14 バイトの動的変数領域を定義するときは、レベルを 2 にして

```
ENTER 14, 2
```

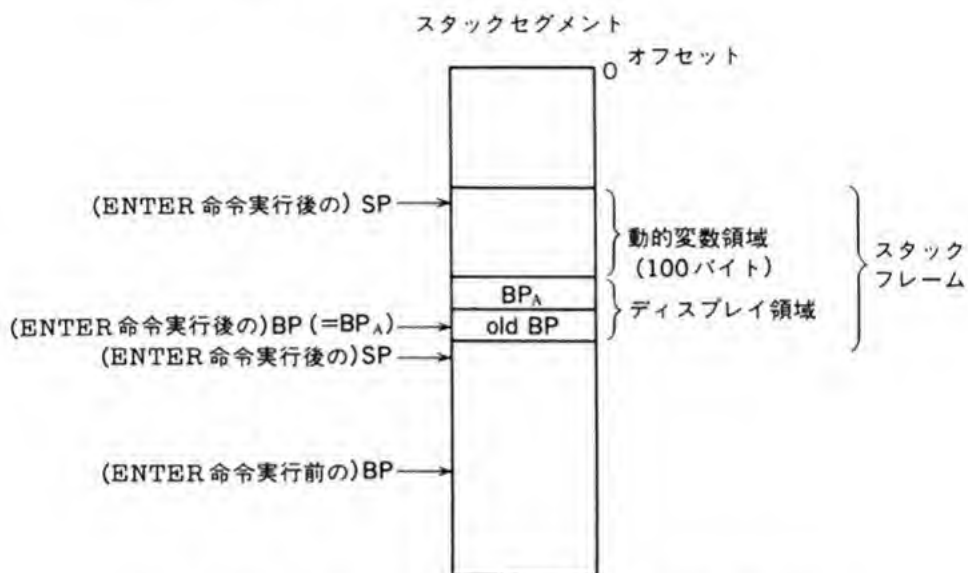


図 2・23 ENTER 命令によって作られるスタックフレーム

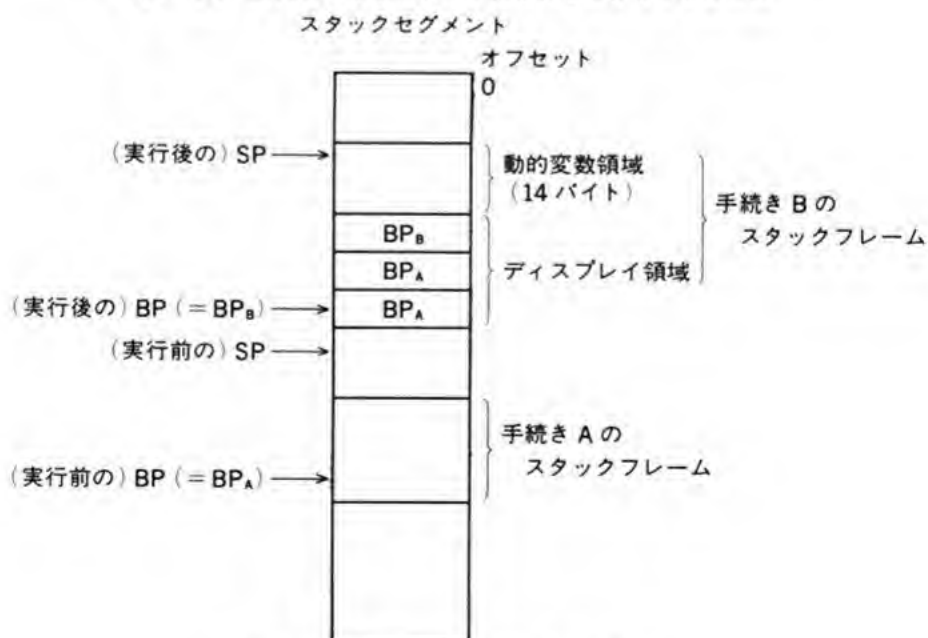


図 2・24 ENTER 14, 2 の実行

のように書く。このとき、図2・21に示した ENTER 命令の動作からわかるように、図2・24に示すようなスタックフレームが作られる。

さらに、図2・22に示した手続き C で実行する ENTER 命令は、レベルを3にすれば、そのスタックフレームのディスプレイ領域に、手続き A、手続き B の



図 2・25 LEAVE 命令の動作

```

I EQU WORD PTR [BP-2]
J EQU WORD PTR [BP-4]
K EQU WORD PTR [BP-6]

PROC_A PROC

    ENTER 6,0

    IN AX,0
    MOV I,AX
    IN AX,2
    MOV J,AX

    ADD AX,I
    MOV K,AX

    OUT 4,AX

    LEAVE
    RET

PROC_A ENDP

```

注) EQU は EQU の右側の定義に、左側に示すようなシンボルを定義する疑似命令である。なお、この手続きの処理自体には意味はあまりない。

図 2・26 ENTER 命令、LEAVE 命令の使用

## 2 リアルモードでの使用

スタックフレームの BP の値  $BP_A$ ,  $BP_B$  を記録することができる。

もちろん、高級言語のすべてが、ENTER 命令のレベルを 1 以上にするような処理を必要としているわけではない。PL/M 言語の動的変数を作るリエントラントな手続きはネストすることが禁じられている。このような場合、レベルを 0 にして

```
ENTER 6,0
```

のように使用してもよい。このとき、ENTER 命令の実行は

```
PUSH BP
```

```
MOV BP,SP
```

```
SUB SP,6
```

の処理と同じである。

ENTER 命令で作成したスタックフレームは、図 2・25 に示す LEAVE 命令を実行して削除することができる。LEAVE 命令はオペランドをもたず、RET 命令の上には書けばよい。このように、ENTER 命令、LEAVE 命令は常に対で使用し、たとえば 3 ワードの動的変数を使用する手続きの定義は、レベルを 0 にするとアセンブリ言語で図 2・26 のように書くことができる。

## 3. プロテクトモードでの使用

80286 の本来の能力はプロテクトモードにおいて発揮される。リアルモードの場合とプロテクトモードの場合では、メモリ管理の方法がまったく異なることを本章で学ぶ。それでもなお、応用プログラムを書くプログラマには、8086 と 80286 の違いは見わけられないだろう。リアルモードの命令は、プロテクトモードでもそのまま使用できる。しかし、ポインタを直接扱うプログラムは、リアルモードとプロテクトモードの間で互換性がなくなることに注意しなければならない。本章では、プロテクトモードでのメモリ管理について説明した後、リアルモードからプロテクトモードへ変換するプログラムを示す。

## 3-1 セグメントキャッシュ

〔1〕 **プロテクトモードの論理アドレス** プロテクトモードにおいてもメモリのアドレスは、論理アドレスによって表し、論理アドレスはセグメントセクタとオフセットからなる。オフセットは、リアルモードと同様にメモリ参照の状況によって IP, SP または命令のオペランドで指定される。また、セグメントセクタも CS, DS, ES, SS の 4 種類のセグメントレジスタに入っている値が使用される。図 3・1 に再びセグメントレジスタとセグメントキャッシュを示すが、プロテクトモードにおいては、このセグメントキャッシュの扱い方が、リアルモードの場合とは異なる。

リミットフィールドには、リアルモードの場合は常に、OFFFHH の値が設定されたのに対して、プロテクトモードの場合は、1 から OFFFHH までの任意の最大オフセット値を設定することができる。リアルモードの場合のベースアドレスは、セグメントセクタを 4 ビットだけ左シフトした値（すなわち 16 倍した値）が設定されたが、プロテクトモードでは、ベースアドレスの値は任意の 24 ビットの値を定義することができる。ここで、セグメントセクタは、セグメントのベースアドレスとは直接に関係のない、セグメントを論理的に識別するためのインデックスとなる。さらに、プロテクトモードでは、アクセスライト (AR) フィールドの 1 バイトの値によって、図 3・2 のような形式で、セグメントの属性を定義することができる。

以上のように、セグメントキャッシュによって表される 6 バイトのデータをセ

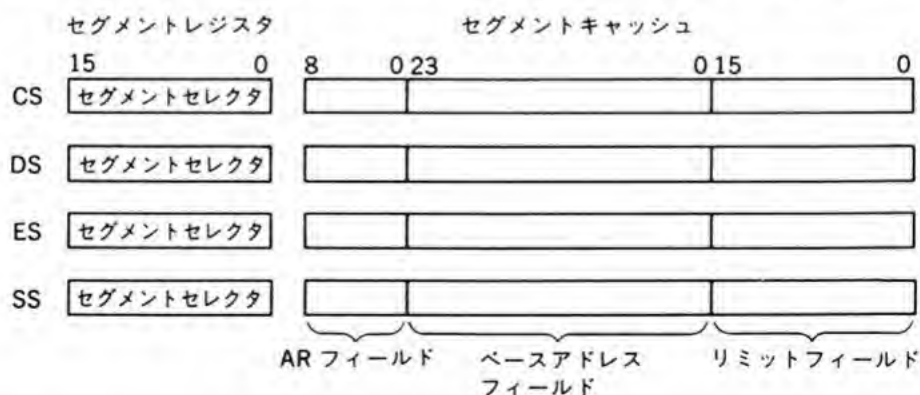


図 3・1 セグメントレジスタとセグメントキャッシュ

### 3-1 セグメントキャッシュ

セグメントディスクリプタと呼ぶ。80286 は、メモリを参照するときセグメントディスクリプタを使用して、オフセットが定義されたセグメントリミットを超えていないかどうか、メモリ参照の方式がアクセスライトに定義した規則にかなっているかどうか、などを検査する。もし、それらの検査の中で1つでも、セグメントディスクリプタに定義した規則に違反していれば、メモリ保護のために 80286 はその命令を中断して、例外割り込みを発生する。命令の実行がすべての検査をパ

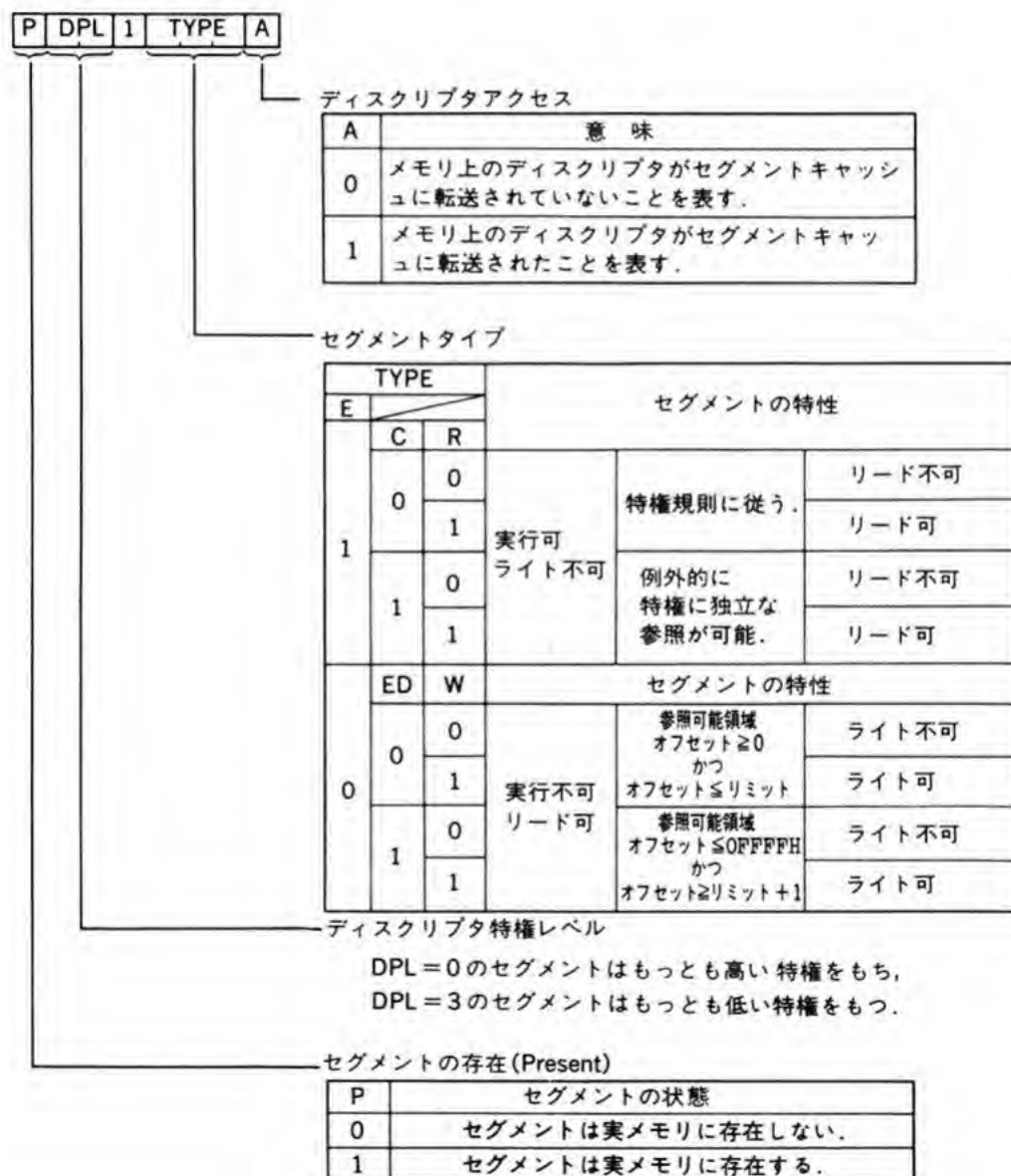


図 3-2 セグメントキャッシュのアクセスバイト

### 3 プロテクトモードでの使用

スしたときにだけ、セグメントキャッシュのベースアドレスにオフセットを加えた物理アドレスをアドレスバスに出力し、バスサイクルを実行する。

セグメントキャッシュに定義するセグメントディスクリプタは、80286 がメモリ参照の正当性を評価するための重要な評価基準である。したがって、誤まったセグメントディスクリプタがセグメントキャッシュに定義されたのでは何にもならない。したがって、セグメントキャッシュには直接に値を代入したり、また逆に、読み出したりできないようになっている。セグメントキャッシュの値は、80286 がその正当性を検査してから設定される。それでは、どのようにしてセグメントキャッシュの値が決まるのであろうか。

#### プロテクトモードの学習

80286 のプロテクトモードは、マルチタスクのオペレーティングシステムを設計するうえで基本的な機能を、CPU のハードウェアまたはマイクロコードに広く応用できる形でもたせたといえることができる。このような 80286 の各機能を学習する場合、とにかく 80286 をプロテクトモードにして多くのテストプログラムを実行させてみるのが一番よい。しかし、初心者にとって困難なことは、80286 のプロテクトモードの動作をひととおり知っていなければ、リアルモードからプロテクトモードに切り換えて 80286 をうまく動作させることが難しい要素があることである。この問題は、リアルモード、プロテクトモードの両方で利用できるモニタプログラムを提供し、初心者にとって最初は困難な処理をモニタプログラムの中に作り込んでおくことによって解決できる。もっといいのは 80286 と同等の機能をソフトウェアで実現したシミュレータを準備することである。このようにすれば 80286 のマシンを持たない人でも 80286 の学習をすることができる。

インテルから 80286 のモニタ SDM 286 およびシミュレータ SIM 286 が提供されているが、現在では 80286 を採用した多くのパーソナルコンピュータが各社から出ているのだから、ビジネス用の OS とは別にプロテクトモードの機能をユーザがテストできるようなモニタなども提供されていいような気がするのだが…。

## 3-2 ディスクリプタテーブル

〔1〕 ディスクリプタテーブル セグメントレジスタは、最も最近に使用されたセグメントを表すと考えることができる。新しいセグメントを参照する場合、4種類のセグメントレジスタのうち、適当なレジスタに目的のセグメントを選択するセグメントセクタを初期設定する。リアルモードのセグメントキャッシュにおいて、実質的に意味のあるデータはベースアドレスだけであり、他のアクセスライト、リミットの値は固定である。そして、セグメントセクタとベースアドレスは

$$\text{ベースアドレス} = \text{セグメントセクタ} \times 16$$

の関係をもち、セグメントの識別はセグメントセクタで表現されたベースアドレスによって行われる。これに対してプロテクトモードでは、アクセスバイト、ベースアドレス、リミットのセグメントディスクリプタの3つの要素によってセグメントを柔軟に定義できる。そのために、OS、ユーザプログラムを含めたシステムで使用するすべてのセグメントに対するセグメントディスクリプタを管理しな

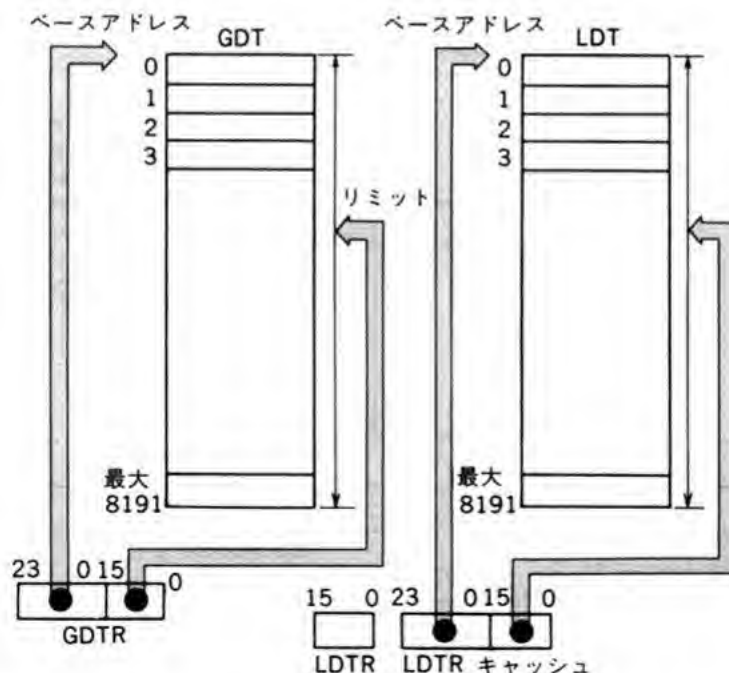


図 3-3 GDT と LDT

### 3 プロテクトモードでの使用



図 3・4 セグメントディスクリプタ

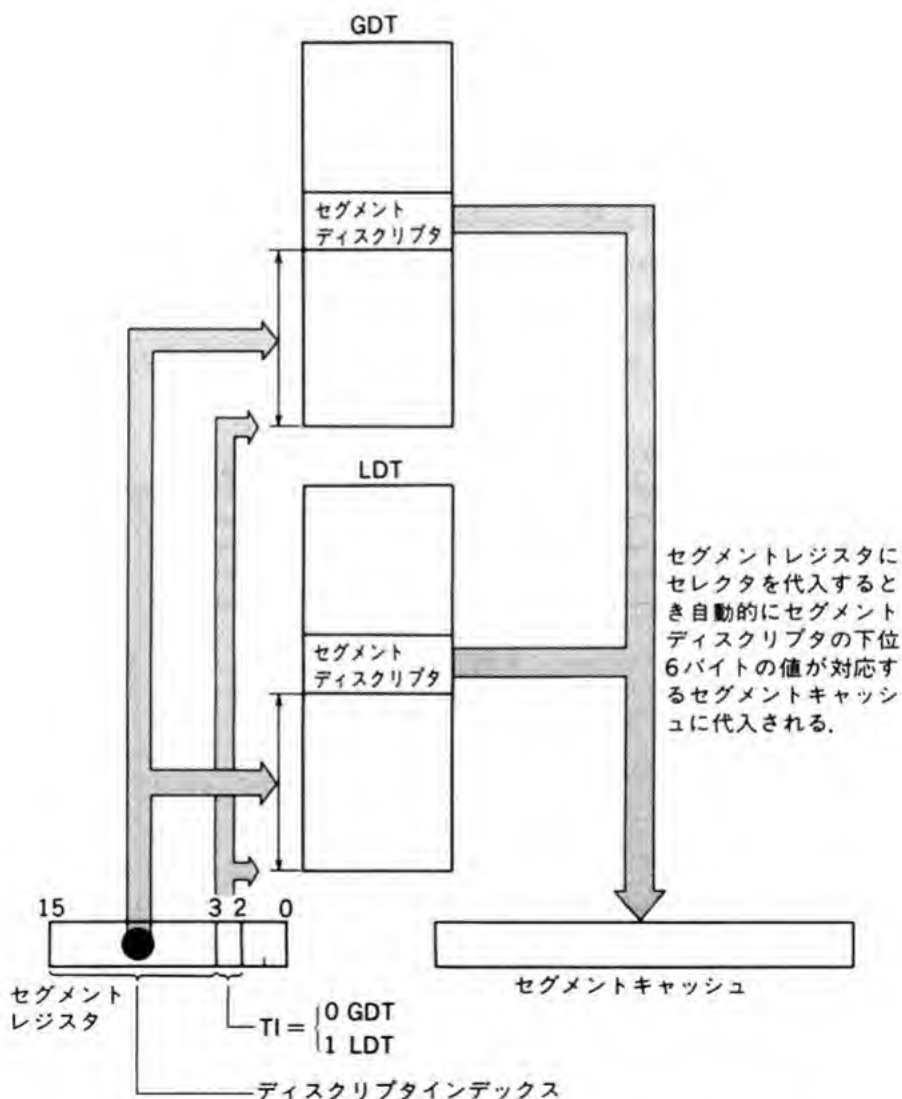


図 3・5 セグメントレジスタとセグメントキャッシュへの代入

ければならない。

80286 は、図 3・3 に示すような **GDT** (**グローバルディスクリプタテーブル**)、**LDT** (**ローカルディスクリプタテーブル**) と呼ばれるメモリに定義された 2 つのテーブルによって、ディスクリプタを管理することができる。GDT、LDT はともに、最大が 64 K バイトの大きさを持ち、図 3・4 に示す形式のセグメントディスクリプタをそれぞれ最大 8191 まで定義することができる (ディスクリプタテーブルのスロット 0 は使用できないので 8192 ではない)。80286 のセグメントディスクリプタは、6 バイトの大きさであるが、ディスクリプタテーブルに定義するときは、図 3・4 に示すように 8 バイトを定義する。8 バイトのセグメントディスクリプタの上位 2 バイトは、32 ビットのマイクロプロセッサ **80386** のディスクリプタと互換性を保つために 0 を代入しておく (80286 プロテクトモードのプログラムは、80386 でもそのまま走らせることができる)。

〔2〕 **セグメントセクタとディスクリプタテーブル** このように、プロテクトモードでは、すべてのセグメントに対応するセグメントディスクリプタを、GDT、LDT のどちらかに定義しておかなければならない。そして、プロテクトモードのセグメントセクタは、ディスクリプタテーブルの中から、1 つのディスクリプタを選択するために使用される。図 3・5 に示すように、セグメントセクタのビット 2 (TI、テーブルインジケータ) が GDT か LDT の識別に使用される。TI が 0 のとき、GDT が選択され、TI が 1 のとき、LDT が選択される。セグメントセクタの上位 13 ビットが、ディスクリプタテーブルのインデックス (スロット番号) を指定する。すなわち、プロテクトモードでは、セグメントセクタによって、1 つのセグメントディスクリプタが決まり、そのセグメントディスクリプタのデータによって、セグメントのベースアドレス、大きさ、その他の属性が決まる。このセグメントセクタをセグメントレジスタに代入するとき、自動的に対応するセグメントキャッシュに、セグメントディスクリプタの下位 6 バイトが代入される。

〔3〕 **ディスクリプタテーブルの定義** したがって、80286 をプロテクトモードで動作させるためには、プロテクトモードに移る前に、GDT、LDT (少なくとも GDT) にセグメントディスクリプタの初期設定をしなければならない。GDT のベースアドレスとリミットは、図 3・6 に示すような GDTR によって定義できる。

### 3 プロテクトモードでの使用

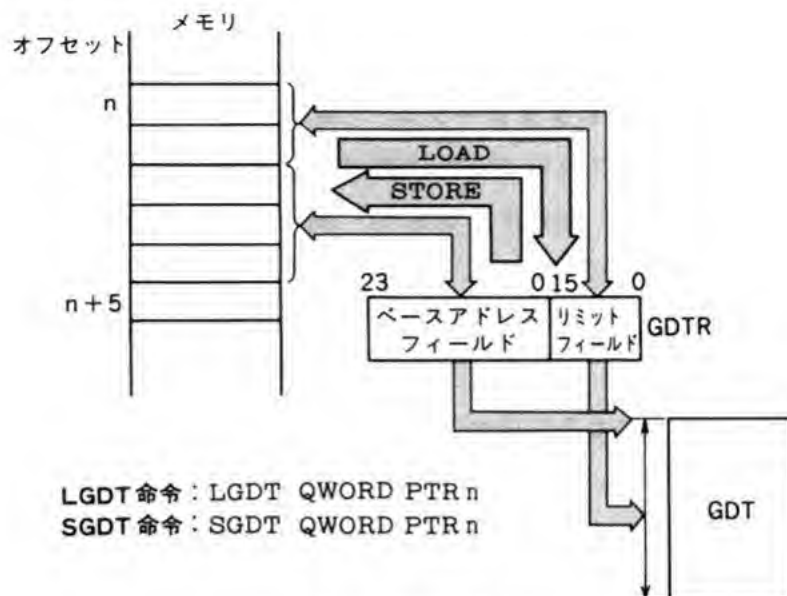


図 3・6 GDT の定義

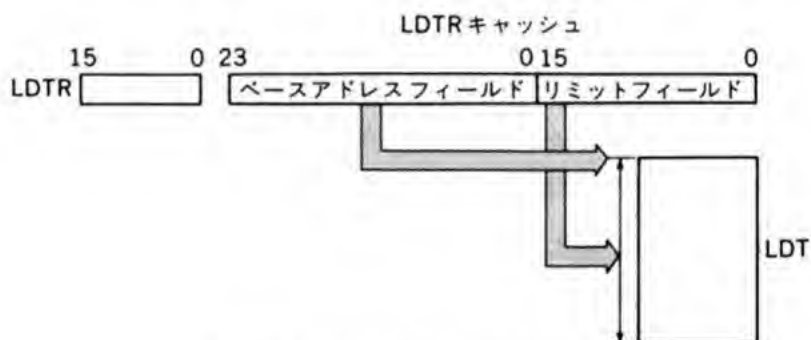


図 3・7 LDTR と LDTR キャッシュ



図 3・8 LDT ディスクリプタ

### 3-2 ディスクリプタテーブル

GDTR は、40 ビットのレジスタで、16 ビットのリミットフィールドと 24 ビットのベースアドレスフィールドからなる。GDTR の値の設定、読み出しは、図 3-6 に示す LGDT 命令、SGDT 命令を使用して実行できる。LGDT 命令はメモリに定義した 6 バイトのデータを GDTR に代入する命令である。このとき、GDTR は 5 バイトの大きさであるから、メモリの 6 バイトのデータの最上位の 1 バイトは無視される。また、SGDT 命令を使用して、GDTR の内容をメモリに書くこともできる。プロテクトモードに移る前に、少なくとも GDT を定義しなければならないので、LGDT 命令、SGDT 命令はともに、リアルモードでも使用することができる。

一方、LDT のベースアドレスとリミットは、図 3-7 に示すように LDTR によって定義されるが、LDTR は GDTR とは形式が異なり、むしろセグメントレジスタのように、キャッシュレジスタをもつ。LDTR キャッシュの大きさは 40 ビットで 16 ビットのリミットフィールドと 23 ビットのベースアドレスフィールドから構成される。

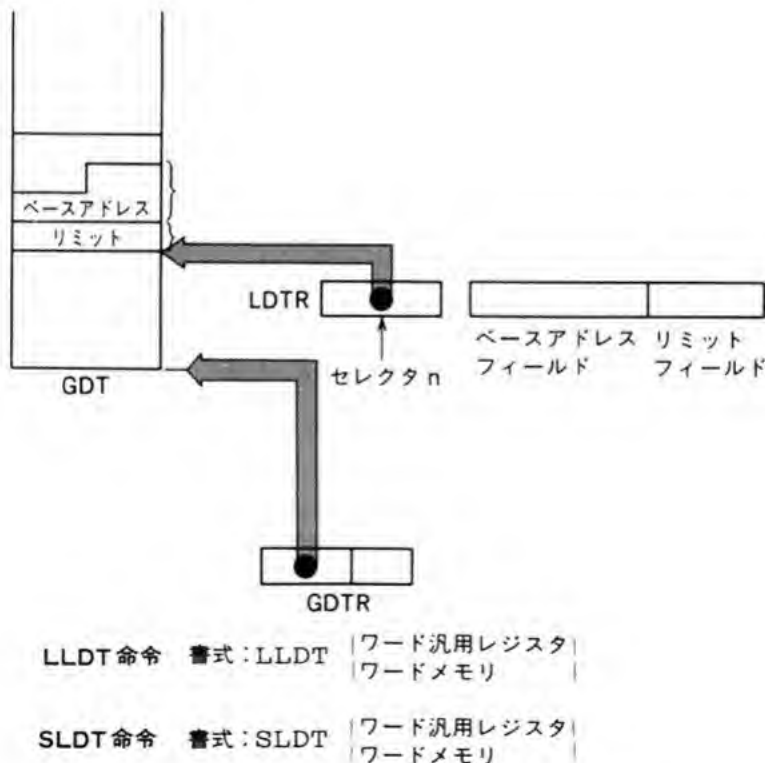


図 3-9 LLDT 命令と LDTR キャッシュの設定

### 3 プロテクトモードでの使用

セグメントキャッシュに代入するデータをセグメントディスクリプタと呼び、ディスクリプタテーブルに定義したのと同じように、LDTR キャッシュに代入するデータを **LDT ディスクリプタ** と呼び、図 3・8 に示すような形式をもつ。LDT ディスクリプタの下位 5 バイトが LDT キャッシュに代入される値である。LDT ディスクリプタのアクセスライトは、LDTR キャッシュには代入されない。LDT ディスクリプタは、必ず GDT に設定しておく。

そこで、LDTR にセクタを代入するためには、図 3・9 に示すように、LLDT 命令を使用する。オペランドに指定したワード汎用レジスタ、またはワードメモリから、LDT ディスクリプタを選択するセクタを LDTR に代入すると、自動的に LDT ディスクリプタの下位 5 バイトが LDTR キャッシュに代入される。逆に、SLDT 命令を使用して、LDTR の値をオペランドに指定したワード汎用レジスタ、ワードメモリに代入することもできる。このときは、LDTR キャッシュの値は変化しない。また、LDTR キャッシュの値を直接に読むことはできない。なお、LLDT 命令、SLDT 命令はプロテクトモードにおいてのみ使用可能である。

このように、GDTR と LDTR のレジスタの形式が異なるのは、GDTR は一度定義してしまえば後に変更することはほとんどないのに対して、LDTR はシステムの状況に応じて、変更することがあるためである。

## 3-3 セグメントレジスタの保護

セグメントレジスタにセグメントセクタを代入することによって、GDT または LDT からセグメントキャッシュにディスクリプタが代入される。セグメントレジスタに値を代入する命令をまとめると表 3-1 のようになる。しかし、80286 が実行する高機能なメモリ管理は、すべてセグメントキャッシュのディスクリプタを評価基準として行われるからセグメントキャッシュに誤まった値が代入されては意味がない。したがって、80286 は、セグメントレジスタ、セグメントキャッシュの値を変更するとき、代入する値の正当性をチェックしている。

〔1〕 DS, ES の保護      たとえば、図 3-10 に示すような GDT が定義されているときに

MOV AX, 101000B      ①

MOV DS, AX      ②

のような操作で、DS にセクタ 101000B を代入することを考える。②の命令において DS, DS キャッシュに値が代入されるまでに、次のような処理が自動的に実行される。

(1) セグメントセクタが 0 であるかどうか検査する。もし、AX の値が 0 であれば DS に 0 を代入し、DS キャッシュが不正な値をもつことを意味するマークを内部的に付けてから命令を終了する。0 のセグメントセクタはヌルセクタと呼ばれ、どのセグメントをも指し示さないことを意味する。したがって、ディスクリプタテーブルのスロット 0 は使用されない。AX の値が 0 でなければ、次の検査に移る。

(2) AX のセクタによって、指定されるセグメントディスクリプタがディスクリプタテーブルのリミット内に定義されているかどうか検査する。もし、指定したセグメントディスクリプタがテーブルリミットを超えているときは、処理を中断してタイプ 13 の割り込みに入る。このとき、DS, DS キャッシュの値は変化しない。セグメントディスクリプタがテーブルリミットを超えていなければ、次の検査に移る。

(3) ここで、初めてディスクリプタの値をテーブルから読む。しかし、無条件にそのまま DS キャッシュに代入するのではなく、まずアクセスライトの値を

### 3 プロテクトモードでの使用

表 3・1 セグメントレジスタを書き換える命令

セグメントレジスタ	セグメントレジスタにセクタを代入する命令
DS, ES, SS を書き換える命令	MOV, LDS, LES, POP
CS を書き換える命令	far タイプ JMP, far タイプ CALL 割り込み

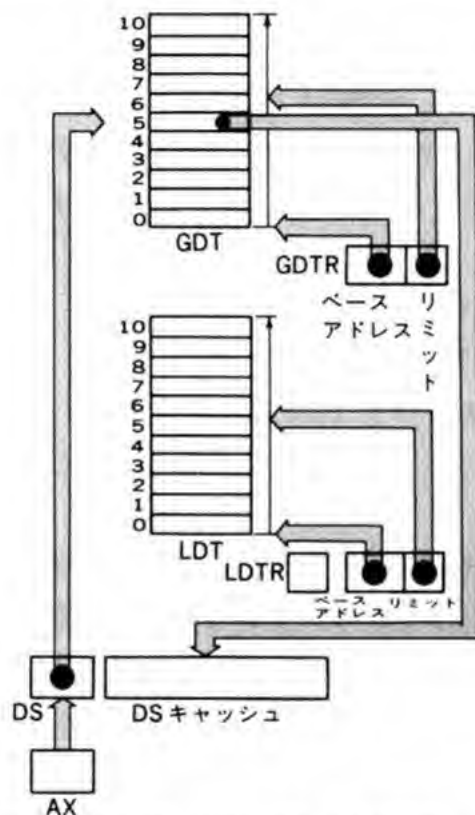


図 3・10 MOV DS, AX の実行

調べる．ここで、このセグメントがデータとして参照することが許されているかどうかを調べる．すなわち、E が 0 であるか 1 であるかである．もし、E が 1 であれば、R が 1 であるかどうかを調べる．もし、このセグメントが、書くことも読むことさえも許されていないならば、80286 は処理を中断して、タイプ 13 の割り込みを発生する．E が 0 であるか、また E が 1 であっても R が 1 であれば、次の検査に移る (図 3・2 参照)．

(4) 最後に、P が 1 であるかどうかを検査する．もし、P が 0 であれば、デ

### 3-3 セグメントレジスタの保護

ィスクリプタは定義されていても、実際のセグメントは実メモリに存在しないことを意味するから、80286 は処理を中断して、タイプ 11 の割り込みを発生する。P が 1 であれば次の処理に移る。

(5) 以上の検査をすべてパスした後、80286 は DS に 101000B を代入し、対応する GDT (5) に定義されているディスクリプタを DS キャッシュに代入する。また、このとき 80286 は自動的に GDT (5) に定義されているディスクリプタの A を 1 にする。

(1) から (5) までの、DS に関する扱いは、ES についてもまったく同様である。

[2] **SS の保護** 次に SS の場合について考える。たとえば

```
MOV AX,1000000B
```

```
MOV SS,AX
```

の処理によって、SS に 1000000B を代入する場合について考える。

SS も DS, ES と同様にセグメントセレクタを代入することができるが、SS はスタックセグメントを指定するという性格上、SS および SS キャッシュに設定できる値は DS, ES の場合と少し異なる部分がある。

(1) SS に代入するセグメントセレクタが 0 かどうかを検査するのは、DS, ES の場合と同様であるが、セレクタが 0 のときは、SS に 0 を代入しないでタイプ 13 の割り込みを発生する。このとき、SS, SS キャッシュの値は変化しない。スタックセグメントは、CALL 命令、RET 命令、割り込み、IRET 命令において自動的に使用されるから、SS には 0 を代入して、無効にすることは許されていない。セレクタが 0 でなければ次の検査に移る。

(2) DS, ES の場合とまったく同様に、セレクタに対応するディスクリプタがディスクリプタテーブルの中にあるかどうかを検査する。

(3) (2) の検査に合格した後、やはりディスクリプタを読み、アクセスライトの値を検査するが、SS に代入する場合はリード、ライトが両方とも可能でなければならない。すなわち、アクセスライトのビットが、E=0, W=1 でなければならない。もし、そうでなければ、タイプ 13 の割り込みが発生する。

(4) 最後に、やはり P が 1 かどうかを検査する。もし、P=0 であればタイプ 12 の割り込みが発生する。

(1) から (4) の検査をすべてパスして、初めて SS に AX のセレクタが代入され、

### 3 プロテクトモードでの使用

SS キャッシュに対応するディスクリプタが代入される。このときも、セグメントキャッシュにディスクリプタが代入されたことを表すため、80286 は自動的にメモリのディスクリプタの A に 1 を書く。

〔3〕 **CS の保護** CS は far CALL, far JMP, 割り込みによって、IP と同時に変更される。たとえば

```
JPTR DD 680000H
```

```
⋮
```

```
JMP JPTR
```

の処理は間接 JMP を使用して、セグメントが 68H、オフセットが 0 の命令に制御を変えるもので、このとき、IP に 0 が代入されるのと同時に、CS に 68H が新しく代入される。この場合も、CS に代入されるセクタに対応するディスクリプタは次のように検査される。

(1) CS に代入されるセクタが 0 かどうかを検査する。もし、セクタが 0 であれば処理を中断し、タイプ 13 の割り込みに入る。CS の場合も SS のときと同様に 80286 の実行において、CS を 0 にして無効にするような状況はありえないから、CS に 0 を代入することは許されていない。

(2) セクタに対応するディスクリプタが、ディスクリプタテーブルのリミット内に定義されているかどうかを検査する。もし、対応するディスクリプタがリミットを超えているものであればタイプ 13 の割り込みを発生する。

(3) 上述の検査に合格したならばディスクリプタを読み、アクセスライトを調べ実行可能かどうかを検査する。もし、E が 0 で実行できないデータが定義されているセグメントである場合、タイプ 13 の割り込みを発生する。

表 3・2 セグメントタイプの検査

セグメント レジスタ アクセス ライト	S(ARのビット 4)=1(セグメントディスクリプタであること)			
	E=0(データセグメント)		E=1(コードセグメント)	
	W=0(ライト不可)	W=1(ライト可)	R=0(リード不可)	R=1(リード可)
DS	○	○	×	○
ES	○	○	×	○
SS	×	○	×	×
CS	×	×	○	○

(4) 次に、P が 1 かどうかを調べ、P が 0 であればタイプ 11 の割り込みが発生する。

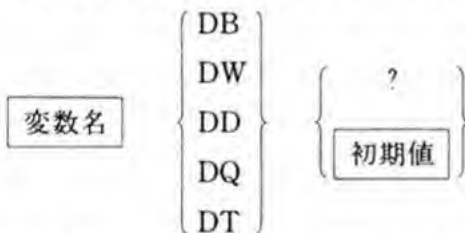
(5) CS の場合は、さらに新しく IP に代入されるオフセットが、新しく CS キャッシュに定義されるディスクリプタのセグメントリミットに入っているかどうかを検査する。もし、セグメントリミットを超えているときは、タイプ 13 の割り込みが発生する。

以上の検査にすべて合格して、初めて 80286 は CS、CS キャッシュおよび IP に新しい値を代入し、異なるセグメントの命令に制御を移行することができる。

このように、セグメントレジスタにはセグメントディスクリプタのタイプによって代入が許されるものと、許されないものがある。この関係を表 3・2 にまとめる。

#### アセンブリ言語における変数定義

アセンブリ言語 ASM 86 および ASM 286 のプログラムで変数を定義するには



の形式で定義する。DB、DW、DD、DQ、DT はメモリにそれぞれ 1 バイト、2 バイト、4 バイト、8 バイト、10 バイトの領域を定義することを表す。その右側に ? を書いたとき、初期値を定義しないことを表し、初期値を指定したときはその値がメモリに定義される。変数名は省略してもかまわないが、オフセット 10 に定義した 1 バイトのデータを AL に代入するという命令を書くとき

```
MOV AL, BYTE PTR 10
```

と書くよりは

```
BDATA DB 123
```

のように変数を定義しておき

```
MOV AL, BDATA
```

のように書く方がよい。変数名を使うことによって、データのオフセットとタイプの定義をアセンブラにまかせることができる。

### 3 プロテクトモードでの使用

#### 3-4 メモリ参照の保護

セグメントレジスタとセグメントキャッシュに正しい値が代入された後、80286 はセグメントキャッシュのディスクリプタを使用して、メモリ参照を実行することができる。ただし、DS、ES に 0 を代入したとき、DS キャッシュ、ES キャッシュはそれぞれ無効となり、これらを使用したメモリ参照を実行すれば、タイプ 13 の割り込みを発生する。SS、CS に 0 が代入されていることはありえない。80286 はメモリ参照の正当性を検査する判断基準として、セグメントキャッシュの値を使用する。

セグメントキャッシュのリミットは、セグメントの大きさを定義する。80286 のセグメントの最大値は、 $2^{16}$  (64 K) バイトであるが、ディスクリプタのリミットとメモリ参照の種類によって図 3-11 に示す領域だけが使用可能となる。64 K バイトのセグメント内で使用できない領域には、他のセグメントを定義することもできるし、使用しないままにしておくこともできる。実行可能なコードセグメントは、オフセットが 0 からリミットまでが参照可能となる。したがって、表 3-3 に示すように、コードフェッチにおいて IP は

$$0 \leq IP \leq \text{リミット}$$

の値をもたなければならない。

アクセスライトの E が 0 で、実行できないデータセグメントの場合、ED が 0 のとき、オフセットが 0 からリミットまでが参照可能であり、ED が 1 のとき、オフセットが [リミット+1] から OFFF<sub>FFH</sub> までが参照可能である。一般に、DS、ES で参照するデータセグメントには ED を 0 とし、SS で参照するスタックセグメントには ED を 1 にする。もし、それぞれのメモリ参照において、オフセットが参照可能な範囲に入っていなければ、表 3-3 に示すように割り込みが発生する。

データ参照においては、表 3-4 に示すように使用するセグメントレジスタとアクセスライトの組み合わせによって、許されるものと許されないものがある。SS キャッシュに代入するアクセスライトは、E = 0, W = 1 であることが、SS にセレクタを代入する段階で検査されるから、表 3-4 において、× 印を付けた状況は起こりえない。

### 3-4 メモリ参照の保護

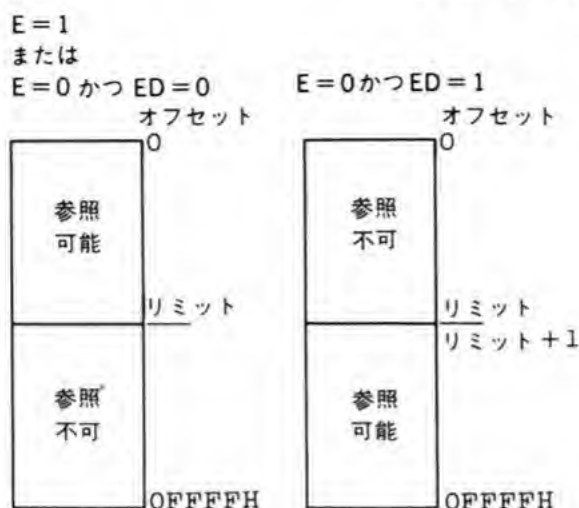


図 3・11 セグメントリミット

表 3・3 リミット検査

メモリ参照	条 件	オフセット検査	例外割り込み
コードフェッチ	E = 1	$0 \leq IP \leq \text{リミット}$	タイプ 13
スタックへの PUSH, POP	E = 0 W = 0 ED = 0	$0 \leq SP \leq \text{リミット}$	タイプ 12
	E = 0 W = 0 ED = 1	$\text{リミット} + 1 \leq SP \leq \text{OFFFHH}$	タイプ 12
データ参照	E = 0 ED = 0	$0 \leq \text{オフセット} \leq \text{リミット}$	タイプ 13
	E = 0 ED = 1	$\text{リミット} + 1 \leq \text{オフセット} \leq \text{OFFFHH}$	タイプ 13

表 3・4 メモリ参照におけるアクセスバイト検査

メモリ参照 / アクセスライト		E = 0		E = 0	
		W = 0	W = 1	R = 0	R = 1
DS, ES を使用	データリード	○	○	例外割り込み タイプ 13	○
	データライト	例外割り込み タイプ 13	○	例外割り込み タイプ 13	例外割り込み タイプ 13
SS を使用	データリード POP	×	○	×	×
	データライト PUSH	×	○	×	×

## 3-5 仮想記憶

〔1〕 **実メモリと仮想メモリ** 80286 は、GDT, LDT の 2 つのディスクリプタテーブルによってセグメントを管理する。セグメントセクタの上位 14 ビットを使用して、GDT, LDT の識別と、テーブル内のディスクリプタの識別を行うから、最大  $2^{14}$  までのセグメントディスクリプタを定義することができる。各ディスクリプタに対応して、最大  $2^{16}$  (64 K) バイトの大きさのセグメントが定義できるから、80286 の論理アドレスでは、**最大  $2^{14} \times 2^{16} = 2^{30}$  (1 G) バイト**までのメモリ空間を表現することができる。

一方、80286 のアドレスバスは 24 ビットであり、ハードウェアに実装できる最大の実メモリは  **$2^{24}$  (16 M) バイト**までである。したがって、論理アドレスで表現できるすべてのメモリ空間を実メモリに入れることはできない。

しかし、図 3・12 に示すように論理アドレス空間を磁気ディスク装置などの高速大容量の補助記憶装置におき、現在必要なセグメントを実メモリに入れるという方法で、OS を作ることができる。このとき、実際には 80286 は最大 16 M バイトまでのメモリをもつことができるが、ユーザは仮想的に 1 G バイトのメモリがあるようにプログラムを書くことができる。したがって、プロテクトモードでは論理アドレスのことを**仮想アドレス**とも呼ぶ。

〔2〕 **仮想記憶の実現** 当然、プログラム実行中に実メモリ上にないセグメントを参照する場合がある。このときは、実メモリ上の未使用領域を探し、もし十分な未使用領域がなければ、現在、実メモリ上にある最も必要でないようなセグメントをディスクに待避し、代わりにディスクから必要なセグメントを実メモリに入れる。このような処理を**メモリスワップ**と呼ぶ。

80286 は、図 3・13 に示すように、アクセスライトの P, A の 2 ビットを使用して、メモリスワップを実現することができる。すなわち、セグメントが実メモリ上に存在するとき、そのアクセスライトの P を 1 にしておく。逆に、セグメントが実メモリ上に存在しないとき P を 0 にしておく。このように、セグメントディスクリプタを定義しておけば、80286 は、P=0 であるディスクリプタをセグメントキャッシュに代入せずに、タイプ 11 の割り込みを発生する(ただし、SS キャッシュに代入するとき、P=0 であればタイプ 12 の割り込みを発生する)。

たとえば、図3・14に示すように、DSにセクタ1BHを代入するとき、対応するディスクリプタのPが0であれば、タイプ11の割り込みを発生する。この割り込み処理において、不必要なセグメントをディスクに待避し、逆に必要なセグメントをディスクから実メモリに代入する。さらに、ディスクリプタのPに1を書いてから、IRET命令によって、割り込みを発生した命令を再実行することができる。図3・14に示すように、メモリスワップの作業は、Pの状態によって80286が自動的に発生する割り込み処理において実現されるから、ユーザはメモリスワップの存在を知らずに仮想アドレスを使用してプログラムを書くことができる。

メモリスワップにおいて、どのセグメントをディスクに待避してどのセグメン

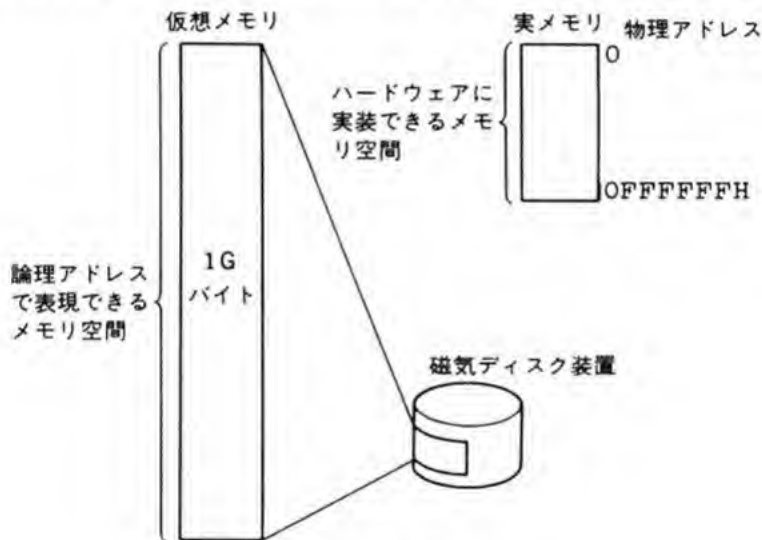


図 3・12 仮想メモリ

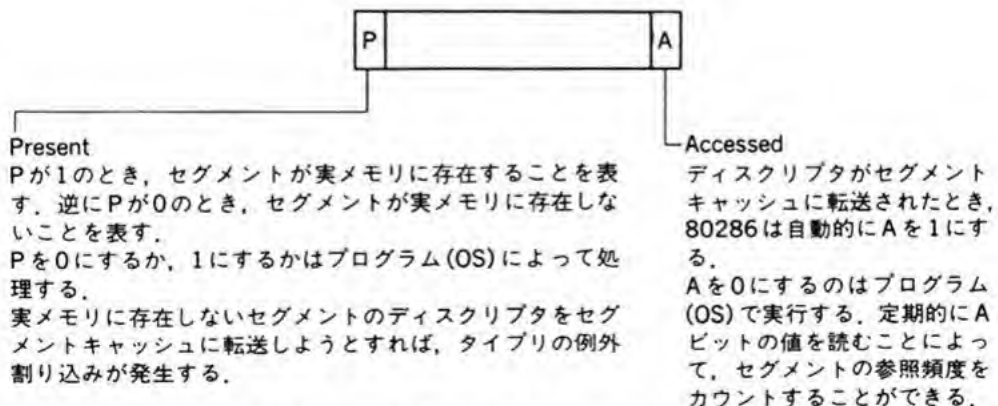


図 3・13 アクセスライトPビット、Aビットによる仮想記憶のサポート

### 3 プロテクトモードでの使用

トをメモリにおくかという管理は OS の中で処理されるが、システムのスピードに影響する重要な作業である。ひんばんに参照されるセグメントは実メモリにおき、参照回数の少ないセグメントをディスクに待避したほうがよい。ここで、アクセスライトの A を使用して、セグメントの参照回数をカウントすることができる。A は 0 に初期設定しておく。80286 はディスクリプタをセグメントキャッシュに代入するとき、自動的に A に 1 を書く。したがって、定期的にディスクリプタの A の値を調べ、その後、A を再び 0 に初期設定するような処理によって、セグメントの参照の回数を計ることができる。

仮想メモリを利用すれば、ユーザは仮想的に 1G バイトまでの非常に大きなメモリ空間を利用することができる。しかし、仮想記憶はディスクと実メモリとの間でのメモリスワップを伴うため、システムの処理速度の低下がある。したがって、仮想記憶は複数のユーザが時分割でシステムを利用するマルチユーザシステムには有効であるが、処理のタイミングが重要なリアルタイムシステムには使用すべきではない。80286 において、仮想記憶を使用しないときは、ディスクリプタの P を常に 1 にしておけばよい。

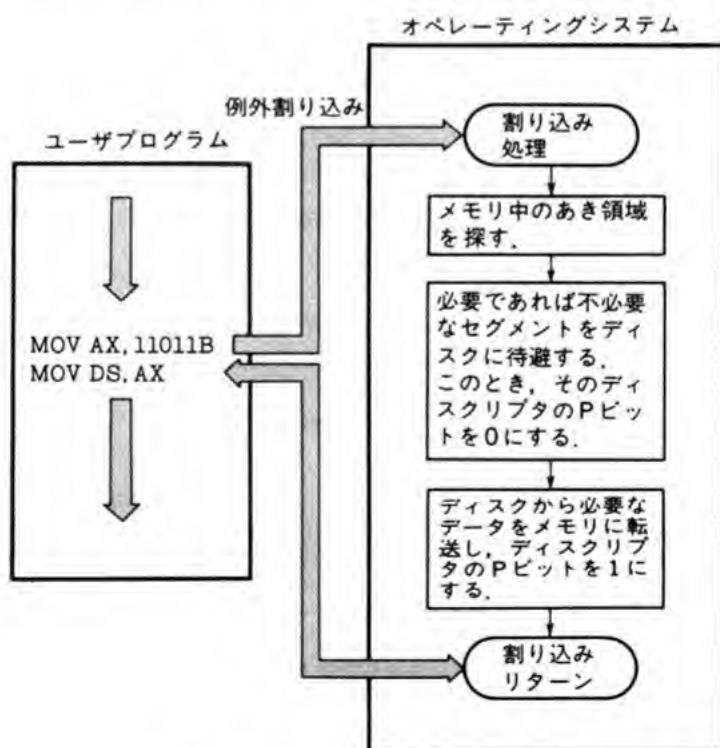


図 3・14 P ビットによる例外処理

## 3-6 ディスクリプタテーブルの扱い

OSの処理では、ディスクリプタテーブルに新しいディスクリプタを追加したり、必要でないディスクリプタを無効にしたり、また、3-5で述べたようにアクセスライトを必要に応じて書き換えることがある。ディスクリプタテーブルを変更する方法は簡単である。ディスクリプタテーブルを1つのデータセグメントとして扱えばよい。たとえば、図3-15に示すようにGDTのスロット1に、GDTをデータセグメントとして扱うためのセグメントディスクリプタをあらかじめ定義しておく。このディスクリプタのアクセスライトをE=0, W=1のように設定しておけば

```
MOV AX, 1000B
MOV ES, AX
```

のように、ESにセクタ8を代入することによって、ESを使用してGDT内のデータを変更することができる。たとえば、図3-15のようにベースアドレス0から始まり、48バイトの大きさをもつGDTがすでに定義されているとすると、

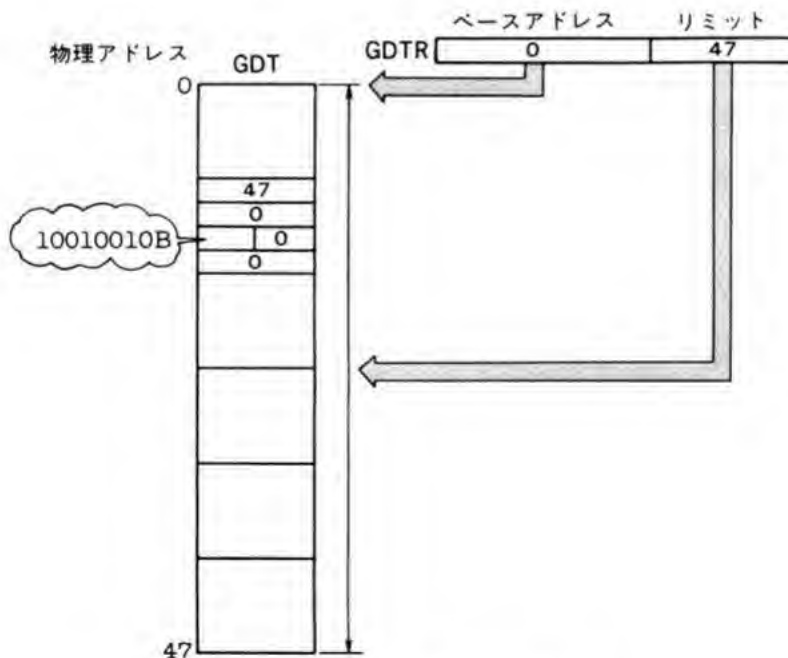


図 3-15 予備ディスクリプタ (alias)

### 3 プロテクトモードでの使用

<b>MOV AX, 1000B</b> <b>MOV ES, AX</b>	⇒ GDT 予備ディスクリプタのセクタ = 8H を ES に代入する。
<b>ADD ES:WORD PTR 8, 8</b>	⇒ ES : 8 の 1 ワードのメモリに 8 を加算する。GDT 予備ディスクリプタのリミットは 47 から 55 になる。
<b>MOV AX, 1000B</b> <b>MOV ES, AX</b>	⇒ ES キャッシュのリミットも 55 にするため再び ES に GDT 予備ディスクリプタのセクタを代入する。
<b>MOV BX, 48</b>	⇒ BX に新しいディスクリプタのオフセットを代入する。
<b>MOV WORD PTR ES:[BX]+0, LIMIT</b>	⇒ 新しいディスクリプタのリミットフィールドに新しいセグメントのリミットを定義する。
<b>MOV WORD PTR ES:[BX]+2, BASEADDRE_LOWER</b>	⇒ 新しいディスクリプタのベースアドレスフィールドの下位ワードに新しいセグメントのベースアドレスの下位ワードを定義する。
<b>MOV WORD PTR ES:[BX]+4, BASEADDRE_UPPER</b>	⇒ 新しいディスクリプタのベースアドレスフィールドの上位バイトに新しいセグメントのベースアドレスの上位バイトを定義する。
<b>MOV WORD PTR ES:[BX]+5, AR_BYTE</b>	⇒ 新しいディスクリプタのアクセスバイトに新しいセグメントのアクセスバイト値を定義する。
<b>MOV WORD PTR ES:[BX]+6, 0</b>	⇒ 新しいディスクリプタの最上位ワードに 80386 との互換性のために 0 を代入する。
<b>LGDT QWORD PTR ES:[BX]</b>	⇒ GDTR に新しい GDT のベースアドレスとリミットを代入する。

図 3・16 GDT の変更処理

GDT のオフセット 48 から始まる新しいディスクリプタを追加するプログラムは図 3・16 のようになる。

一般には、OS を設計する場合、ディスクリプタを必要なたびに新しく作るのではなく、前もって見積もっただけの大きさのディスクリプタテーブルを定義したほうがよい。このとき、使用しないディスクリプタのアクセスライトには 0 を書いておく。アクセスライトが 0 のとき、そのディスクリプタをヌルディスクリプタと呼び、もし間違ってもヌルディスクリプタをセグメントキャッシュに代入しようとするれば、80286 は自動的にタイプ 13 の割り込みを発生するので、システムの誤まりを発見することができる。

## 3-7 プロテクトモードの初期設定

以上見たように、80286 をプロテクトモードで動作させるためには、メモリに少なくとも GDT を定義し、GDT のベースアドレス、リミットを GDTR に代入しなければならない。これらの処理は 80286 がリアルモードで動作中に実行する必要がある。GDT を定義する方法は 2 つある。

1 つの方法は、あらかじめ ROM に書き込んだ GDT を RAM にコピーする。または、ディスクファイルに定義した GDT の初期データをメモリに代入する方法がある。

ここでは、図 3・17 に示すように物理アドレス 0FF000H から 4K バイトの大きさの GDT の初期データが ROM に書き込まれているとして、物理アドレス

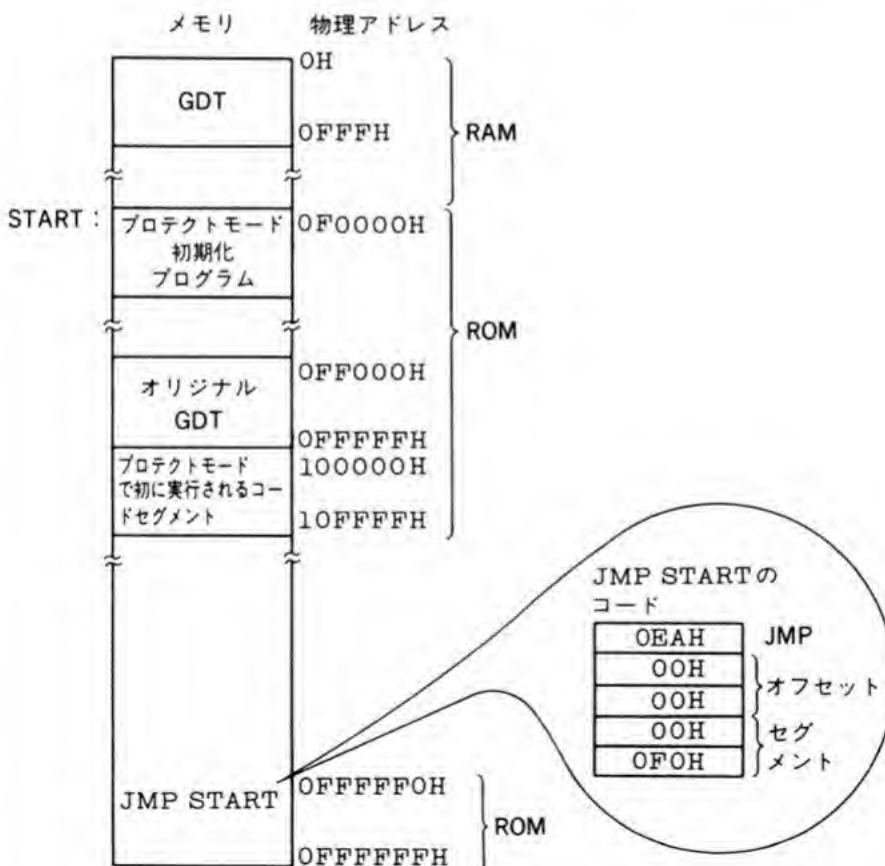


図 3・17 GDT の初期設定

### 3 プロテクトモードでの使用

0H から始まる 4K バイトの大きさのメモリに GDT を定義し、プロテクトモードに切り換えてから、物理アドレス 10000H から始まるプログラムを実行するまでの処理の流れについて考える。ここで、物理アドレス 0FF000H から始まる ROM には、図 3・18 に示すような GDT の初期データが定義されているものとする。図において、オフセット 8 から始まる 8 バイトの領域に GDT の補助ディスクリプタが、オフセット 24 から始まる 8 バイトの領域にプロテクトモードで最初に実行されるコードセグメントのディスクリプタが定義されている。

80286 をリセットしたとき、リアルモードで物理アドレス 0FFFFFF0H にある命令を最初に実行する。ここには一般に図 3・17 に示したように、最初に実行

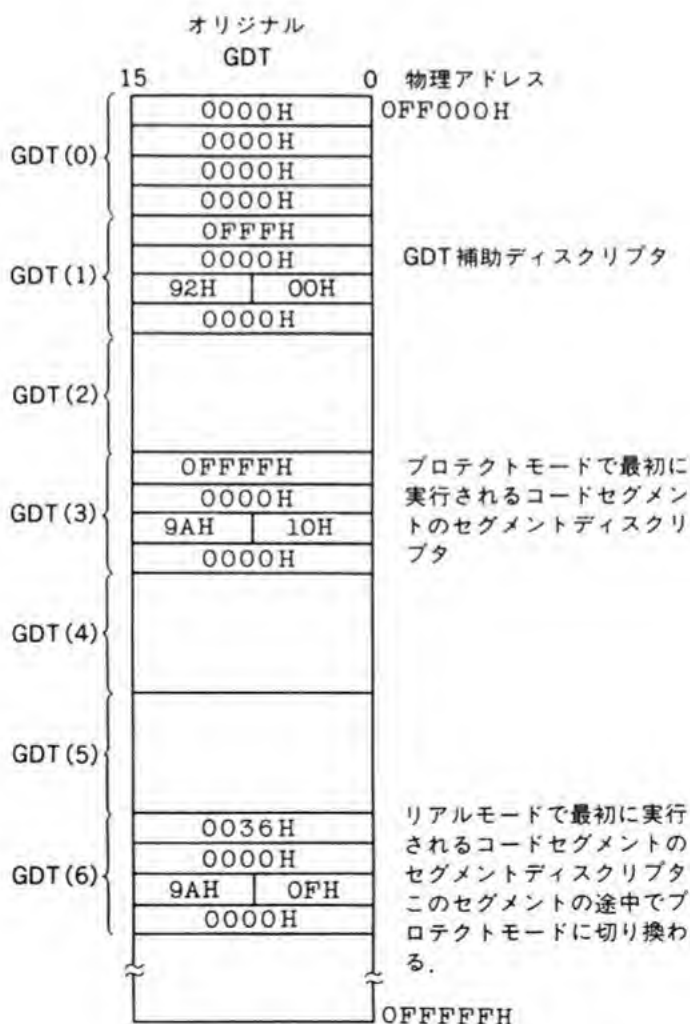


図 3・18 ROM 化してオリジナル GDT

### 3-7 プロテクトモードの初期設定

されるプログラムに制御を移す JMP 命令をおく。この JMP 命令で、セグメントセクタが 0F000H、オフセットが 0 のアドレスへジャンプし、図 3-19 に示すプロテクトモード初期化プログラムを実行する。このとき、アドレスバスと A<sub>20</sub>-A<sub>23</sub> は 0FH から 0 に変わることには注意する。

ここでは、GDT の初期化だけについて考えたが、プロテクトモードにスイッチする前に IDT の初期化も行う必要がある。IDT については第 6 章において述べる。また、LDT の初期化はプロテクトモードになってから実行する。なぜなら、LGDT 命令はリアルモードでも、プロテクトモードでも使用できるのに対して、LLDT 命令はプロテクトモードでしか使用できないからである。



図 3-19 プロテクトモード初期化プログラム



## 4. 特権レベルによる保護

マイクロコンピュータシステムのソフトウェアは、大きく分類して OS (オペレーティングシステム) または **スーパーバイザ** と呼ばれる管理プログラムと、その下で働くさまざまな **応用プログラム** (ユーザープログラム) によって構成されている。このようなシステムにおいて、1つのユーザープログラムが暴走しても、OS の処理に影響しないようにするためにはどうすればよいだろうか。また、第3章で見たように GDT, LDT のディスクリプタを書き換えることは、システムの動作に致命的な影響を及ぼす。ユーザープログラムが GDT, LDT を勝手に書き換えないようにするためにはどうすればよいだろうか。80286 では各セグメントに特権レベルを付けて階層化し、一部の不完全なプログラムによって OS などのシステムに重要な処理またはデータが壊されるのを防ぐことができる。

## 4 特権レベルによる保護

### 4-1 特権レベル

〔1〕 OSの保護 OSは、ファイル管理、メモリ管理、タスク管理などのプログラムによって構成される。ユーザプログラムからは必要に応じて、OSのもつこれらの機能をサブルーチンコール（手続きの引用）という形で利用することができる。簡単のために、図4・1に示すようなシステムを考える。OSの中にFREADという手続きが定義されている。FREADは、ユーザプログラムからファイルの識別データ、メモリのポインタなどをパラメータとして受け取り、ファイルから入力したデータをユーザプログラムのバッファ領域に転送する処理である。

図4・1に示す例から、OSとユーザプログラムの間には一般に次のような関係が成り立つ。

（1） ユーザプログラムから、OS内部のサブルーチンへパラメータを渡してコールできなければならない。しかし、ユーザプログラムの暴走によって誤ってOS内部に制御が移らないように、制御の移行を検査できる機能が必要である。また、サブルーチンリターン以外に、OSからユーザプログラムに制御を移行する必要はない。なぜならば、OSレベルのプログラムが、OS内部のプログラムより信頼性の低いと思われるユーザ定義の手続きを引用することはないからである。

（2） OS内部のプログラムからユーザプログラム内部のデータセグメントは自由に参照できなければならない。逆に、ユーザプログラムがOS内部のデータセグメントを参照することは禁止するべきである。

このように、ユーザプログラムに対して、OSだけに特権を与えることによって信頼性の高いシステムを作ることができる。特権は**特権レベル**と呼ばれる番号によって区別する。OSを設計する場合、図4・1の例に示したように、特権レベルは2レベルあればなんとか間に合う。UNIXまたはXENIXがカーネルとユーザプログラムの2レベルの特権を使用している例である。しかし、OS内部においても、カーネルまたはニュークリアスと呼ばれるOSの最も中心となる部分と、デバイスドライバなどの部分を特権によって分離したい場合がある。さらに、データベースシステムのような本来のOSには属さないが、ユーザプログラムよ

## 4-1 特 権 レ ベ ル

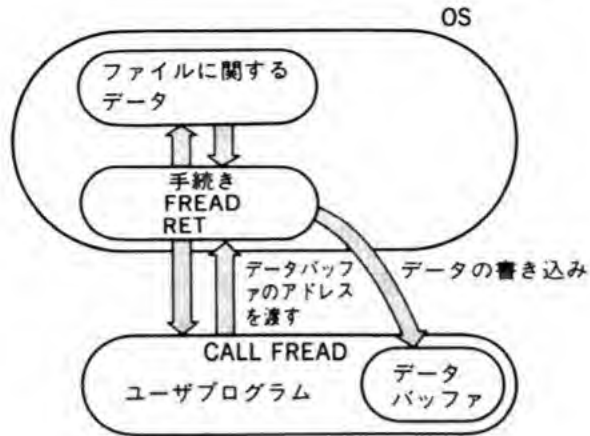


図 4・1 ユーザプログラムと OS



セグメントの特権レベルはセグメントディスクリプタのアクセスライト中のDPLによって定義される。

図 4・2 セグメントの特権レベルの定義

GDTで管理されるセグメント群      LDTで管理されるセグメント群

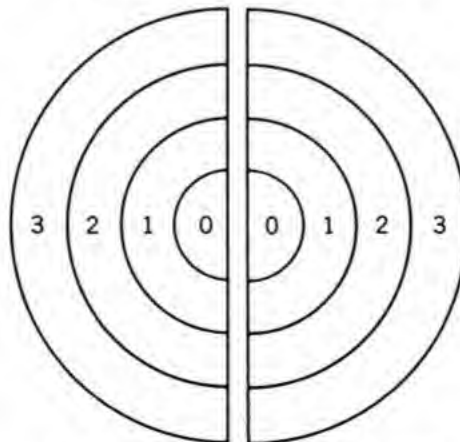


図 4・3 特権レベルによるセグメント管理

#### 4 特権レベルによる保護

りも上の特権に設定したいプログラム群もある。

〔2〕 80286 の特権保護 80286 では、上述のようなあらゆる種類の OS 設計に柔軟に対処できるように、4 レベルの特権をセグメントごとに定義することができる。セグメントの特権レベルは、図 4・2 に示すようにセグメントディスクリプタの **DPL** (descriptor privilege level) の 2 ビットに、0 から 3 までの数値で定義する。数値的に最も小さい特権レベル 0 が最も大きな特権をもち、数値的に最も大きな特権レベル 3 が最も小さい特権をもつ。特権レベルを考慮すれば、80286 の論理アドレス空間は図 4・3 に示すように表現することができる。

特権レベルの利用は、図 4・4 に示すように OS の最も中心となるカーネルを特権レベル 0 に定義する。これは、LGDT 命令、LLDT 命令など、80286 の動作に根本的に影響する命令は特権命令と呼ばれ、特権レベル 0 でしか使用できない

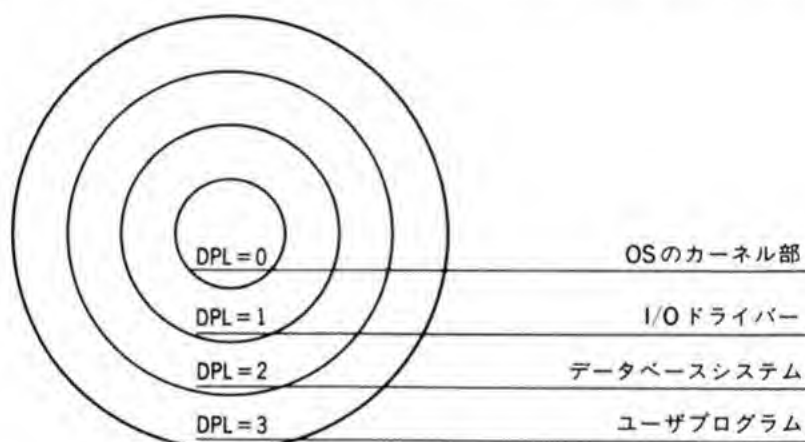


図 4・4 80286 で定義できる特権レベル

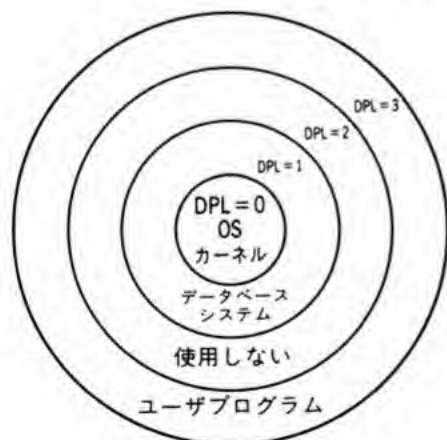


図 4・5 OS 設計に必要な特権レベル

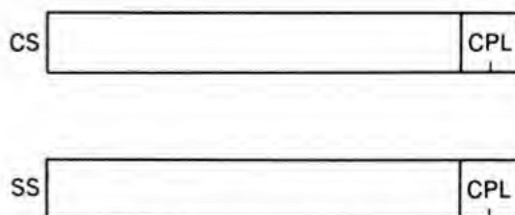


図 4・6 CPL (現在の特権レベル)

からである。その他のプログラムは特権レベル 0 以外の特権を任意に利用してよいが、一般にユーザプログラムは特権レベル 3 に定義する。図 4・4 に示すように、特権レベル 1 に I/O ドライバ、特権レベル 2 にデータベースシステムというように定義してもよい。しかし、必要もなくすべての特権レベルを使用するべきではない。図 4・5 に示すように、デバイスドライバをカーネルと同じ特権レベル 0 に定義し、データベースシステムを特権レベル 1 に、ユーザプログラムを特権レベル 3 に、それぞれ定義してもよい。XENIX 286 は特権レベル 0 と 3 を使用し、他の特権レベルは使用していない。また、特権による保護が必要なければ、すべてのセグメントを特権レベル 0 に定義すればよい。

80286 は図 4・6 に示すように、CS、SS の下位 2 ビットとディスクリプタの DPL を比較することによって、特権による保護を実行する。CS、SS の下位 2 ビットは常に同じ値であり、**CPL** (現在の特権レベル) と呼ぶ。80286 が特権による保護を実行するとき、CPL の値を特権保護の評価基準として使用する。

#### メモリ管理の方法

メモリ管理とは、プログラムが使用してよい領域といけない領域を識別できる機能である。このために、セグメンテーションとページングの 2 つの方法がある。

セグメンテーションは、コード領域、データ領域、スタック領域などのセグメントと呼ばれる論理的単位でメモリの部分集合を扱う。ディスクリプタテーブルと呼ばれる特別なテーブルによって各セグメントのアドレス、大きさなどが管理される。ページングは、記録されるデータの論理的意味に関係なく、ページと呼ばれる一定の大きさ (たとえば 4 K バイト) の単位にメモリを分割して管理する。

セグメントはその大きさを柔軟に定義できるが、ページは固定長であるから、ページングによってメモリの論理的な管理までもたせることは不利である。しかし、仮想メモリのシステムにおいてメモリとディスクとの間でデータの交換を行う処理には固定長のページは適している。

8086 のメモリはセグメンテーションを採用するが、ディスクリプタテーブルによる管理機能のない限定されたものである。80286 はディスクリプタテーブルによって管理されるセグメンテーションをもつ。そして、80386 は 80286 と同様のセグメンテーションに加えてページング機能をもつ。

## 4-2 データセグメント、スタックセグメントの特権保護

図4-7に示すように、特権レベル  $n$  のデータセグメント D\_SEG に定義された変数 VAR に定数 3 を代入する処理を考える。3つの MOV 命令が、特権レベル  $n$  のコードセグメント C\_SEG、 $n$  より数値的に小さい特権レベル 1 のコードセグメント A\_SEG、そして、 $n$  より数値的に大きい特権レベルのコードセグメント B\_SEG にそれぞれ定義されている。この中で、B\_SEG から変数 VAR を参照することは許されない。これは、4-1 で述べた「ユーザプログラムが OS 内のデータセグメントを参照することは禁止するべきである」という保護を、80286 が特権レベルの値に応じて実行するからである。

たとえば

```
MOV DS, AX
```

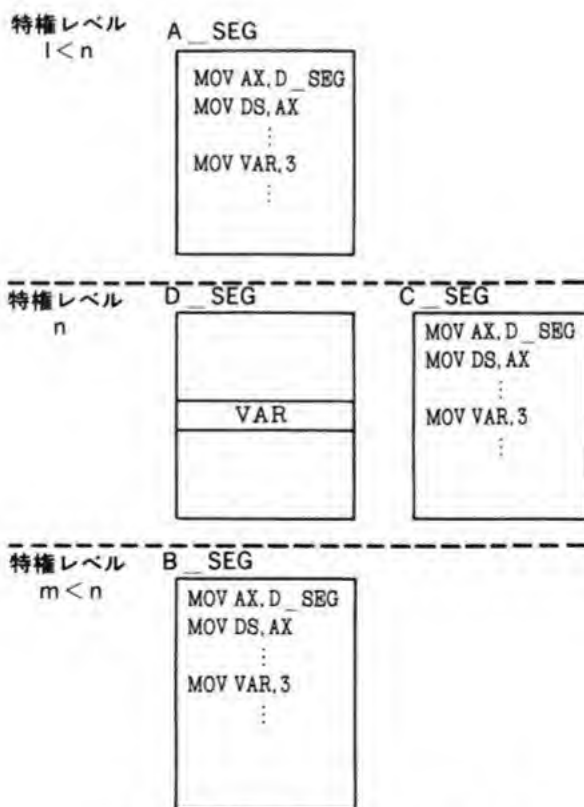


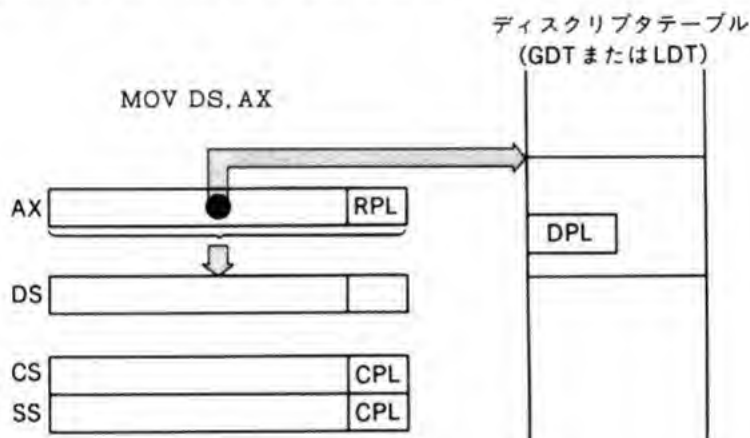
図 4-7 データセグメントの特権保護

## 4-2 データセグメント、スタックセグメントの特権保護

の命令で、DS にセグメントセクタを代入するときの状況を図 4・8 に示す。このとき、3-3 で述べた保護に加えて

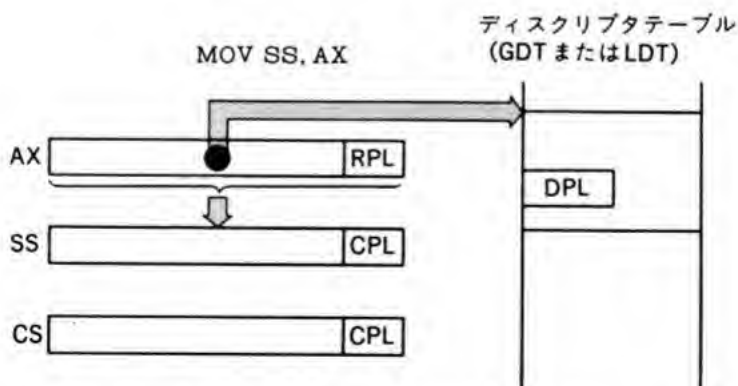
$$DPL \geq \text{MAX}(CPL, RPL)$$

の関係を満足しなければならない。ただし、MAX ( ) は括弧の中のパラメータに指定した最も大きな数値を表すものとする。ここで、CPL は MOV DS, AX を実行する現在の特権レベルを表し、RPL は新しいセクタを DS に代入することを要求するコードセグメント (すなわち、MOV AX, セグメントセクタを実行したコードセグメント) の特権レベルを表す。多くの場合、CPL = RPL と考えられるが、CPL ≠ RPL の場合もある。このとき、80286 自身が実行する



特権について、 $DPL \geq CPL$ かつ $DPL \geq RPL$ の条件を満たさなければならない。

図 4・8 DS または ES の特権保護



特権について、 $DPL = RPL$ かつ $DPL = CPL$ の条件を満たさなければならない。

図 4・9 SS の特権保護

#### 4 特権レベルによる保護

特権保護だけでは完全ではない。しかし、今は  $RPL = CPL$  と考えてよい。RPL と CPL が異なる場合については 4-8 で述べる。上述の条件は、実行中のコードセグメントの特権レベルより大きな特権の（特権レベルが数値的により小さい）セグメントで定義されたデータを参照できないことを表す。もし、この条件が満足されないときは特権保護によって 80286 は、タイプ 13 の割り込みを発生する。このとき DS および DS キャッシュの値は変化しない。

以上のことは ES についても DS の場合とまったく同様である。

次にスタックセグメントの保護について考える。たとえば、図 4-9 に示すように

`MOV SS, AX`

の命令によって、新しいセグメントセレクタを SS に代入する。このとき

$DPL = CPL = RPL$

の関係が成立しなければ、80286 は SS および SS キャッシュの値を変更せずにタイプ 13 の割り込みを発生する。すなわち、スタックセグメントの特権レベルは、常に現在実行中のコードセグメントの特権レベルに等しくなければならない。

## 4-3 コードセグメントの特権保護

CSを書き換える命令には、リアルモードの場合と同じように far JMP, far CALL, そして割り込みがある。割り込みについては第6章で述べるので、ここでは far JMP と far CALL について考える。図4・10に far CALL を使用してコードセグメント CODE 2 から、コードセグメント CODE 1 に定義された手続き PROC 1 に制御を移行する例を示す。CALL PROC 1 の命令を実行するときの状況を図4・11に示す。far CALL は、5 バイトの命令で、制御を移行する先のセグメントセレクタとオフセットをそれぞれ2バイトで表す。オフセットが IP に、またセグメントセレクタが CS に代入されるが、このとき

$$RPL \leq CPL \text{ かつ } CPL = DPL$$

の条件を満足しなければならない。もし、上述の条件を満足しなければ 80286 は CALL 命令を実行せずにタイプ 13 の割り込みを発生する。すなわち、一般の far CALL では、特権レベルの異なるコードセグメントへ制御を移行することは

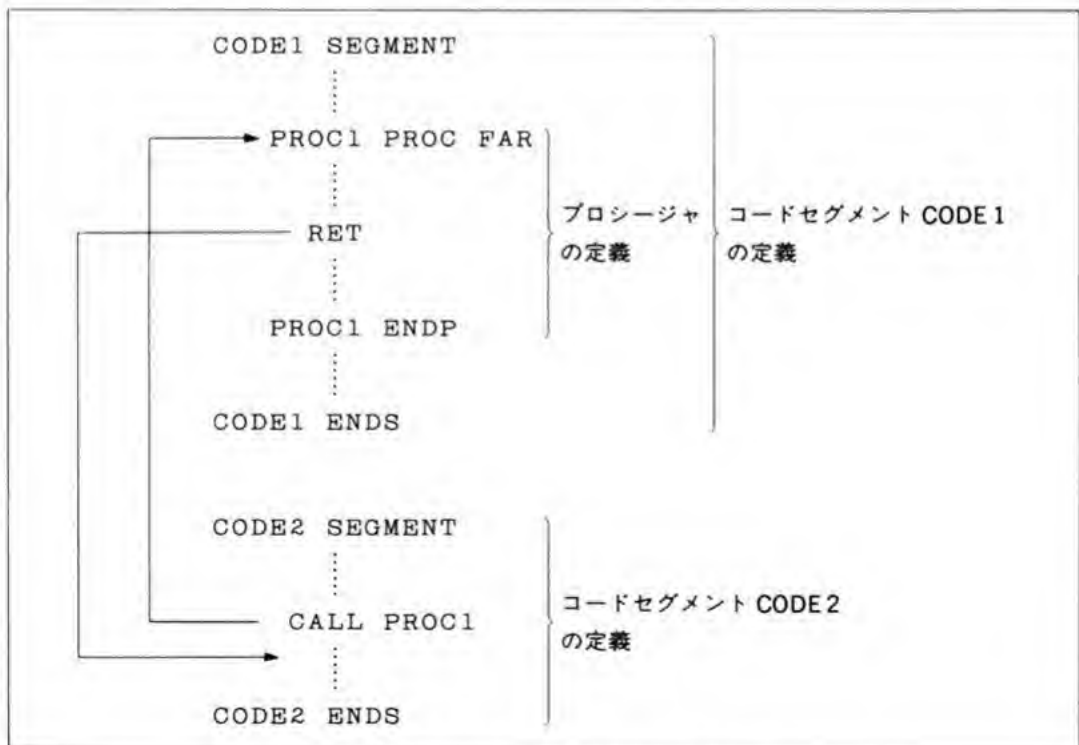
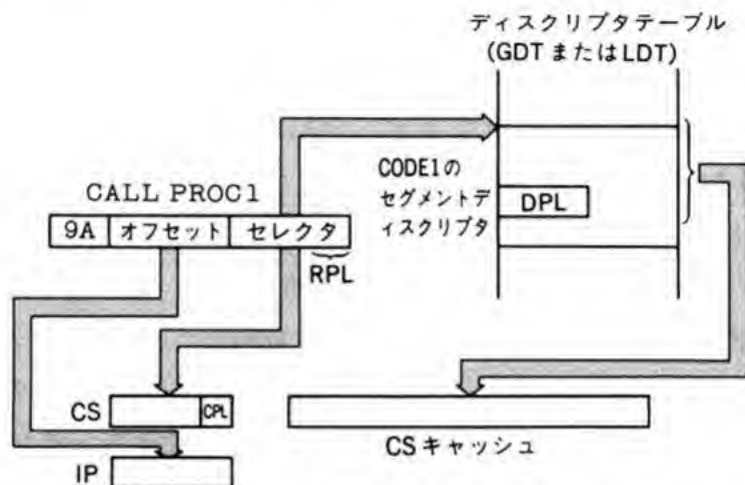


図 4・10 セグメント外への制御移行

#### 4 特権レベルによる保護

できない。したがって、特権の低いプログラムが暴走しても、特権レベルの異なるプログラムに誤って制御が移行することがない。このことは CALL 命令が制御を移行する前に戻りアドレスをスタックに PUSH することを除けば、far JMP 命令についてもまったく同じである。



特権について、 $RPL \leq CPL$ かつ  $CPL = DPL$  の条件を満たさなければならない。

図 4・11 CS の特権保護

## 4-4 コールゲートによる制御移行

CPL を変更して、より高い特権レベルのコードセグメントに制御を移行する場合、**コールゲート**と呼ばれる特殊なディスクリプタを使用する。コールゲートは、セグメントディスクリプタのようにセグメントの属性を定義するものではなく、より大きな特権のコードセグメントで定義された手続きへの制御移行の入口を定義するディスクリプタである。コールゲートは、図 4-12 の形式をもち、GDT または LDT に定義する。プログラムでコールゲートを参照する場合、そのセクタによって識別することができ、far CALL 命令または far JMP 命令のセクタ部 (5 バイトの命令コードの最上位の 2 バイト) にコールゲートのセクタを定義することができる。このとき、far CALL 命令および far JMP 命令は、制御を移行する前にコールゲートを読み、コールゲートに定義されたデータに従って動作する。

図 4-12 に示すように、コールゲートには制御を移行するセグメントセクタとオフセットを下位 2 ワードに定義するが、セグメントセクタの RPL の 2 ビットは使用しない。**ワードカウント**には 0 から 31 の値を指定することができ、ここにはスタックを介して手続きに渡すパラメータのワード数を定義する。コールゲートの P, DPL は、データセグメントに対するものと同じ扱いである。なぜならば、コールゲート自体が CALL 命令、JMP 命令で参照される特殊なデータと考えられるからである。すなわち、コールゲートを参照する far CALL 命令、far JMP 命令は、 $CPL \leq DPL$  となるような CPL のコードセグメントで実行されなければならない、また P は 1 でなければならない。

図 4-13 に、コールゲートを参照する CALL 命令と、それによって CS, IP の値が変わる様子を示す。図において、EXTRN CGATE:FAR は、コールゲート CGATE が他のファイルで定義され、CGATE を参照する命令を far タイプにすることを表す。もし、CGATE が GDT (5) で定義されているとすると、CALL CGATE のオブジェクトコード (機械語) は、9AH, 00H, 00H, 08H, 02H の順にアドレスの小さいほうからメモリに配置される。

この CALL 命令を実行したとき、GDT (5) から読んだディスクリプタがコールゲートであることを 80286 が発見すれば、コールゲートに定義されたオフセッ

#### 4 特権レベルによる保護

トを IP に代入し、またコールゲートに定義されたセグメントセレクタを CS に代入し、さらに CS で指定されるセグメントディスクリプタを CS キャッシュに代入する。このとき、制御を移行するセグメントディスクリプタの DPL に関して

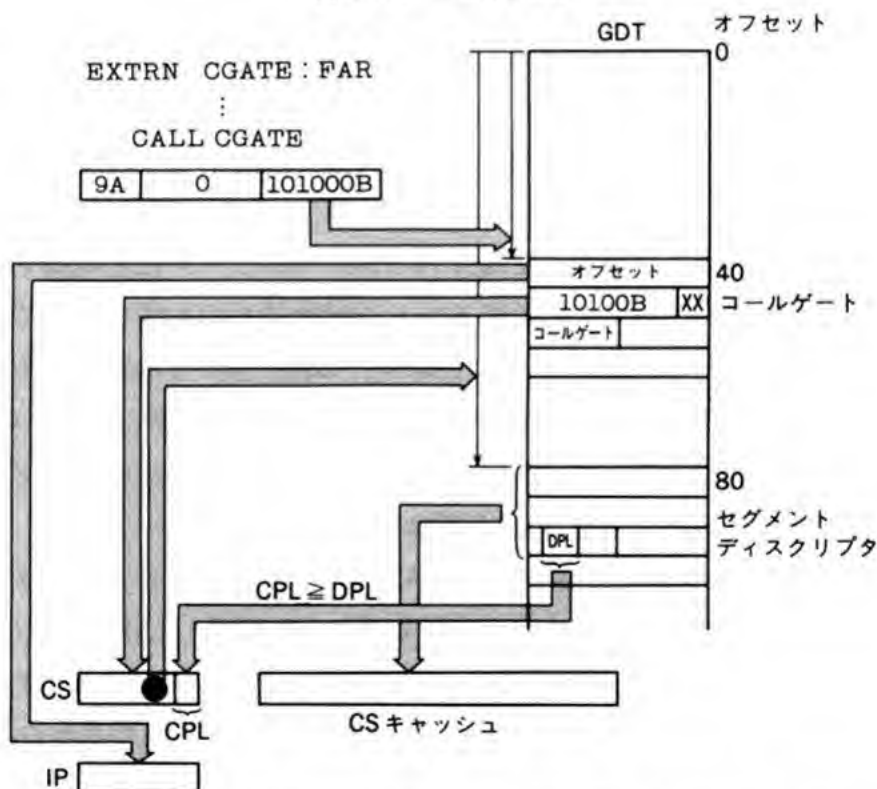
$$CPL \geq DPL$$

が成立するならば、DPL が新しい CPL になる。なお、far CALL のセレクタが

オフセット										+0
セグメントセレクタ								X	X	+2
P	DPL	0	0	1	0	0	X	X	X	+4
80386 のための予約領域 0										+6

X は 80286 がその値を無視することを表す。

図 4・12 コールゲート



注) EXTRN は、その右に指定したシンボルが他のファイルで定義されタイプが FAR タイプであることを表すアセンブリ言語の疑似命令である。このとき、CALL CGATE は far CALL 命令となる。

図 4・13 コールゲートを介した制御移行

#### 4-4 コールゲートによる制御移行

コールゲートを指定する場合は、命令コードの中のオフセットの値は使用されない。このように、コールゲートを参照する CALL 命令によって、CPL を変更してより高い特権レベルのセグメントへ移行することができる。

JMP 命令のオペランドにもコールゲートを指定することはできるが、JMP 命令の場合は、コールゲートを使用してもなお制御を移行する新しいセグメントディスクリプタの DPL に関して

$$CPL = DPL$$

の条件が必要である。したがって、JMP 命令で特権レベルを変えることは不可能で、コールゲートを使用する意味はない。コールゲートを参照する CALL 命令において、特権レベルが変化しなくてもかまわないが、そのような制御移行にわざわざコールゲートを使用するのはあまり意味がない。

#### ASM286 (アセンブラ) のコールゲートの扱い

80286 の命令で、オペランドにコールゲートを指定する特別な CALL 命令、JMP 命令があるわけではない。80286 の CALL 命令、JMP 命令は 8086 のものとなんら変わるところはない。CALL 命令、JMP 命令の参照するセクタがゲートを指していればゲートを介した制御移行となるし、セグメントを直接に指していればゲートを介さない制御移行となる。ASM286 でプログラムするとき、CALL 命令のオペランドが外部で定義されているならば

```
EXTRN OS_ROUTINE:FAR
```

のようにプログラムの先頭で宣言して

```
CALL OS_ROUTINE
```

のように書けばよい。このとき、OS\_ROUTINE がコードセグメントのディスクリプタを指しているのか、コールゲートを指しているのかということを意識する必要はない。

## 4-5 スタックセグメントの保護

コールゲートを参照する CALL 命令で特権レベルを変更したとき、80286 は自動的に SS と SP の値を変更する。なぜならば、特権を切り換えても高い特権レベルのコードセグメントが低い特権レベルのコードセグメントと共通のスタックセグメントを使用していたとすれば、低い特権レベルからの影響がスタックを介して高い特権レベルのプログラムに及ぶからである。したがって、図 4・14 に示すように 0 から 3 までのすべての特権レベルを使用するならば、それぞれの特権レベルにそれぞれ独立なスタックセグメントを定義する。なお、コールゲートを使用した CALL 命令でも、特権レベルが変わらなければスタックセグメントも変わらない。

たとえば

```
PUSH PARA1
PUSH PARA2
PUSH PARA3
CALL CGATE
```

のように、3 ワード (6 バイト) のパラメータをスタックに PUSH してから、図 4・15 に示すようなコールゲート CGATE を介して制御の特権レベル 3 から特権レベル 0 に移行する場合について考える。図 4・16 にスタックセグメントの変更の様子を示す。特権レベルが変わるとき、新しい特権レベルのスタックセグメントの SS, SP の初期値を TSS (task status segment) と呼ばれる特別なセグメントから読み、SS, SP に代入する。この後、前のスタックセグメントの SS, SP の値を新しいスタックに PUSH し、前のスタックから新しいスタックにパラメータをコピーする。このとき、コピーするパラメータのワード数

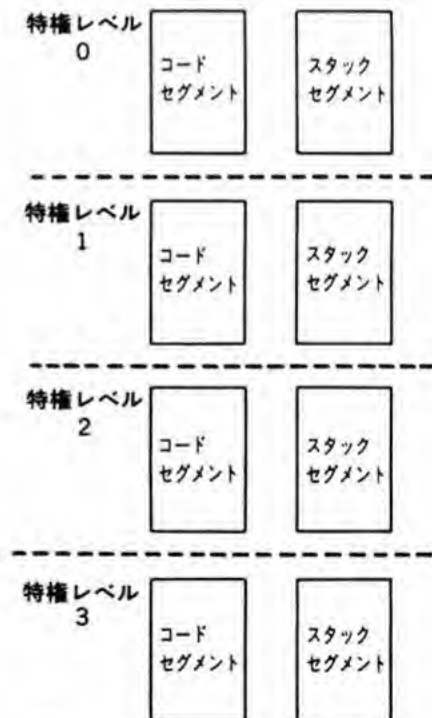


図 4・14 コードセグメントとスタックセグメントの対応

## 4-5 スタックセグメントの保護

は、コールゲートのワードカウントで定義される値である。最後に新しいスタックに戻りアドレスのCSとIPをPUSHする。このように特権レベルが変わるとき、CALL命令のスタック操作は少し複雑であるが図4・16からもわかるように、新しいスタックの状態は8086のプログラムでパラメータをスタックで渡す場合と同じである。

なお、TSSのオフセット2から12までに、0から2までの各特権レベルで使用するSS、SPの初期値が定義されている。コールゲートを使用して、他の特権レベルから特権レベル3に移行することはないから、特権レベル3のSS、SPの初期値の定義は必要ない。TSSについては第6章においてより詳しく説明する。

ディスクリプタテーブル  
GDTまたはLDT

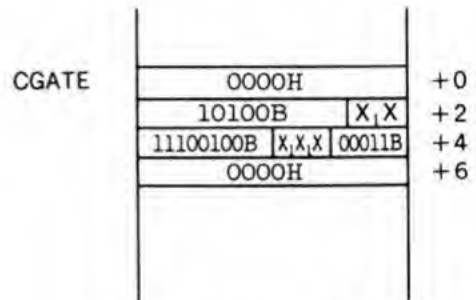


図 4・15 CGATE の定義

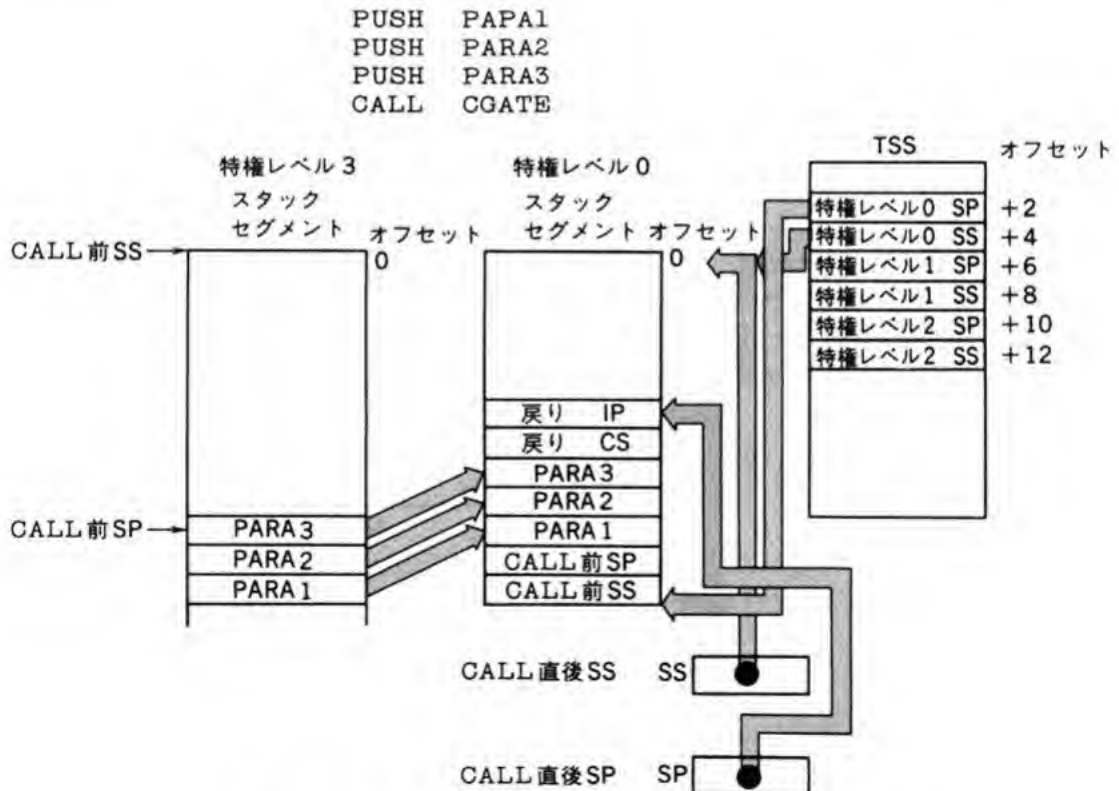


図 4・16 コールゲートによるスタックセグメントの切り換え

## 4-6 RET 命令による制御移行

4-5で述べたように、コールゲートを使用したCALL命令によって特権レベルを変更して、制御を移行した場合にもRET命令によってCALL命令の次の命令に制御を戻すことができる。このときの様子を図4-17に示す。スタックに6バイトのパラメータがPUSHされている場合のRET命令は

RET 6

のように、オペランドにパラメータのバイト数を指定しなければならない。

まず、スタックにPUSHされている戻りCSのRPLとCPLが比較され、

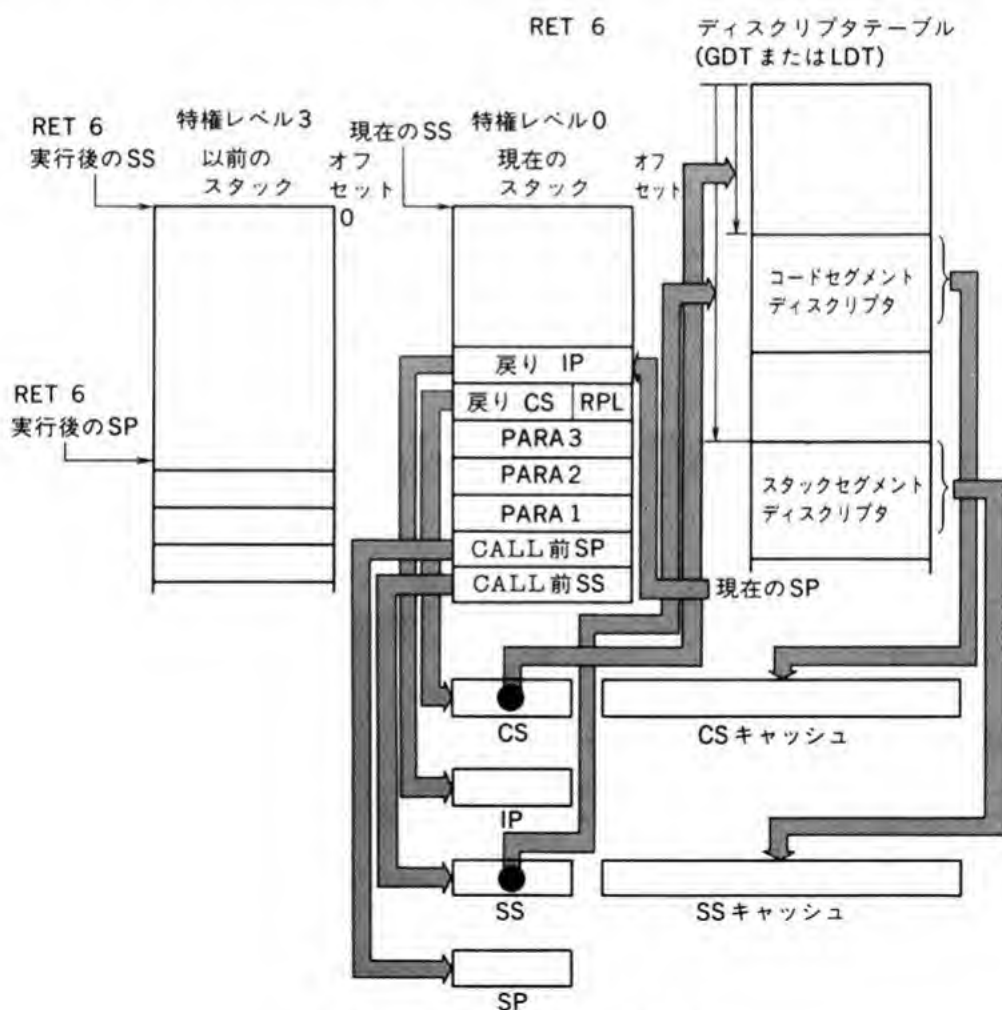


図 4-17 RET 命令による制御移行

RPL と CPL が等しい場合はリアルモードのときと同じ RET 命令が実行される。また RPL が CPL より数値的に大きな場合は、CALL 命令のときと逆に特権の低いコードセグメントへ戻る。このとき、SS、SP の値も自動的に [現在の SP+4+RET 命令のオペランド] のオフセットから保存されている、CALL 前の SS、SP の値によって書き換えられる。また、CS、SS が変更されたとき、CS キャッシュ、SS キャッシュも変更され、ディスクリプタの検査も自動的に実行される。

RET 命令によって低い特権レベルに制御が戻るとき、DS、DS キャッシュ、ES、ES キャッシュの値についても検査される。もし、RET 命令で戻るコードセグメントの CPL に対して、DS または ES のキャッシュの DPL が数値的に小さい場合は、80286 は自動的に DS または ES にヌルセクタを代入し、DS、ES を使用できないようにする。

#### 8086, 80286 の RET 命令

8086, 80286 の RET 命令には 2 章で述べたように far RET 命令と near RET 命令の 2 種類がある。

注意すべきことは、far CALL と far RET、near CALL と near RET は必ず対応させて使用しなければならないし、また同じ手続き中の RET は far タイプか near タイプのどちらかに統一しなければならない。

しかし、RET 命令は命令のニーモニックで far と near を区別することはできない。なぜなら、far RET 命令と near RET 命令の使い分けをプログラマにまかせることは、プログラマの負担が大きくなるからである。かわりにアセンブリ言語 ASM 86, ASM 286 では

```
PROC { FAR
      NEAR }
      ...
ENDP
```

のように、手続きを定義する疑似命令をもつ。PROC NEAR (このとき、NEAR は省略可能) と ENDP の間に書かれた RET 命令はすべてアセンブラによって near RET と解釈され、PROC FAR と ENDP の間の RET 命令はすべて far RET と解釈される。

## 4 特権レベルによる保護

### 4-7 コンフォーミングコードセグメント

高い特権レベルのコードセグメント内部で定義された手続き、データを、必要に応じて低い特権レベルから参照したい場合がある。図4・18に示すように、実行可能なコードセグメントのアクセスライトのCを1にしたものは、コンフォーミングコードセグメントと呼び、例外的な特権保護が実行される。

コンフォーミングコードセグメントで定義された手続きに、CALL 命令、JMP 命令を用いて制御を移行する場合

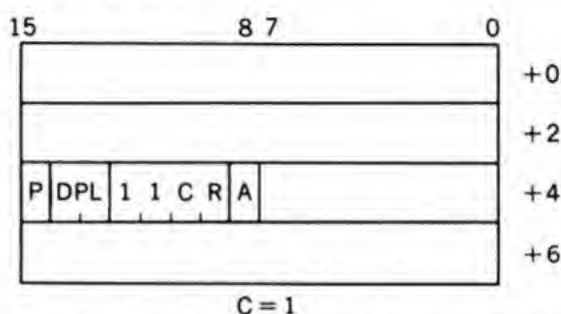


図 4・18 コンフォーミングコードセグメントのアクセスライト

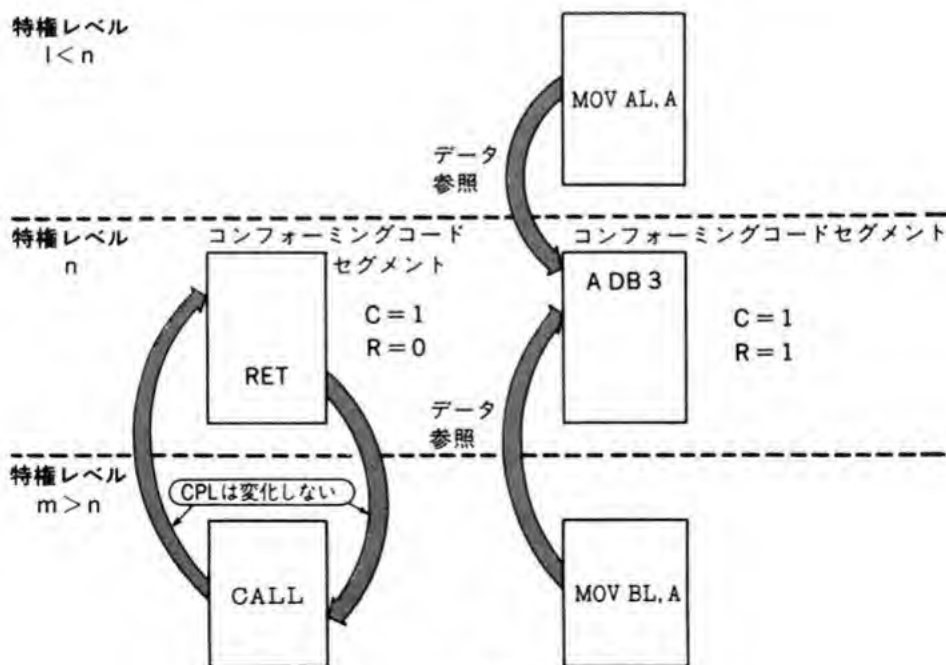


図 4・19 例外特権規則

#### 4-7 コンフォーミングコードセグメント

コンフォーミングコードセグメントの  $DPL \leq CPL$

の条件が満足されればコールゲートを使用する必要はない。すなわち、図 4-19 に示すようにコンフォーミングコードセグメント内部の手続き、ラベルには、特権レベルの低いセグメントからであれば、どこからでも CALL 命令あるいは JMP 命令を使用して制御を移行することができる。ただし、コンフォーミングコードセグメントに制御を移行しても、CPL は変化しない。CALL 命令、JMP 命令を実行したとき、CPL が 3 であれば、 $DPL = 0$  のコンフォーミングコードセグメントで定義された処理を実行する間も、CPL は 3 のままである。したがって、コンフォーミングコードセグメント以外の、特権レベルの高いセグメントを参照することはもちろんできない。

さらに、コンフォーミングコードセグメントのアクセスライトの R が 1 で、リード可能となっている場合、このセグメント内部のデータはすべて、任意の特権レベルからリードすることができる。

#### ASM 86 と ASM 286

8086 から 80386 までの命令は上位互換性がはかられている。8086、80186 のアセンブラは ASM 86 である。80186 では、8086 の命令に加えて 10 種類の新しい命令が含まれるが、この場合は ASM 86 を実行するコマンドに MOD 186 のオプションスイッチを指定する。ASM 286 は 80286 のオブジェクトプログラムを作成するアセンブラである。

オブジェクトプログラムはインテルで定義されたフォーマットでファイルに格納される。ASM 86、ASM 286 も含めて一般にアセンブラが作るオブジェクトファイル化は再配置形式である。再配置形式とは、プログラムをメモリの任意のアドレスに配置できるように、プログラムを配置するアドレスによって影響される部分を変更可能にしたものである。

再配置形式オブジェクトファイルは、プログラムを配置するメモリのアドレスを決め、部分的な再編集をするまで実行することはできない。これらの作業は一般にリンカ、ローケータと呼ばれるユーティリティプログラムによって実行される。

## 4-8 「トロイの木馬」問題

〔1〕 **トロイの木馬** システムの信頼性にとって、セグメントレジスタに代入されるセクタは重要な鍵となる。80286 は特権レベルによって、セグメントレジスタに代入する前にセクタの値を検査し、ユーザプログラムが OS のデータを勝手に読んだり、書き換えたりしないように保護することができる。しかし、セクタを提供するプログラムの特権レベルと、そのセクタをセグメントレジスタに代入するプログラムの特権レベルが異なるとき、この保護は完全ではなくなる。

図 4・20 に簡単な例を示す。OS が特権レベル 0 に定義され、特権レベル 3 のユーザプログラムは OS 内部で定義された手続き COPY を、コールゲート COPY\_GATE を介して使用することができる。手続き COPY はパラメータとして、2つのメモリ領域のポインタと転送するバイト数を受け取り、メモリからメモリへ指定されたバイト数だけのデータを転送する。この簡略化 OS のプログラムリストを図 4・21 に示す。

手続き COPY を使用して、転送処理を実行するユーザプログラムのリストを図 4・22 に示す、このプログラムは、ソース（転送もと）メモリのセクタ、オフセット、デスティネーション（転送先）メモリのセクタ、オフセット、そして転

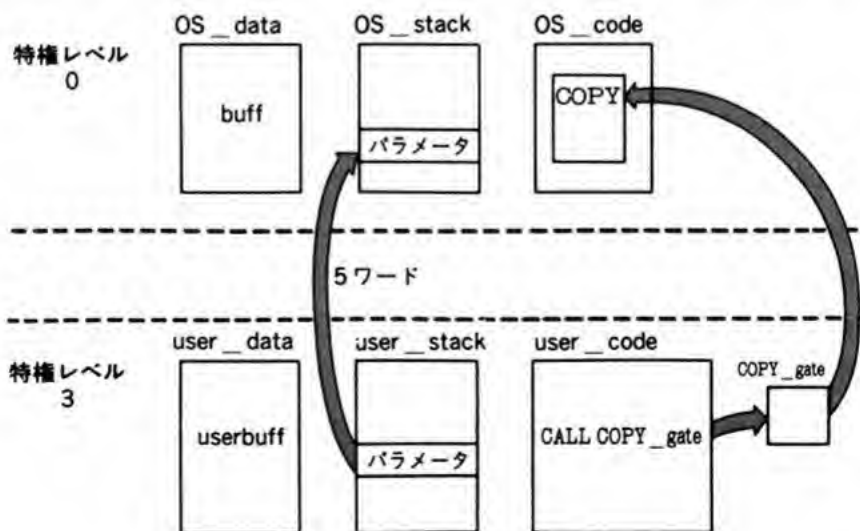


図 4・20 トロイの木馬

```

NAME simplified_os

PUBLIC copy

os_data SEGMENT RW
    buff DB 'THIS DATA SHOULD BE PREVENTED FROM THE OUTER LEVEL.'
os_data ENDS

os_stack STACKSEG 100H

os_code SEGMENT EO
    ASSUME DS:os_data  ➡ ASSUME 宣言はセグメント OS_data 内で定義
                        した変数は DS を使用して参照することをアセン
                        ブラ ASM286 に指定する疑似命令である。
    ASSUME SS:os_stack ➡ 上と同様に、OS_stack 内で定義した変数は SS
                        を使用して参照することを ASM286 に指定する。
                        ASM286 は ASSUME 宣言に従って自動的にセグ
                        メントオーバーライドプリフィックスを作るので、プ
                        ログラマが各命令個々に指定する必要はない。

    stack_format STRUC
        old_bp      DW ?
        return_ptr  DD ?
        count       DW ?
        destin_ptr  DD ?
        source_ptr  DD ?
    stack_format ENDS

    copy PROC FAR
        PUSH BP
        MOV BP,SP
        PUSH DS
        PUSH ES
        PUSH SI
        PUSH DI
        PUSH CX

        LDS SI,[BP].source_ptr
        LES DI,[BP].destin_ptr
        MOV CX,[BP].count
        CLD
        REP MOVSB

        POP CX
        POP DI
        POP SI
        POP ES
        POP DS
        POP BP
        RET 10
    copy ENDP
os_code ENDS
END

```

図 4・21 簡略化 OS の定義

#### 4 特権レベルによる保護

```

NAME userprogram
EXTRN copy_gate:FAR
user_data SEGMENT RW
    userbuff DB 100H DUP(?)

user_data ENDS
user_stack STACKSEG 100H
user_code SEGMENT EO
    ASSUME DS:user_data
    ASSUME SS:user_stack

start:
    PUSH 11000B      ⇒ 転送するデータが定義されているメモリのセグメント
                        セレクタを PUSH する。
    PUSH 0            ⇒ 転送するデータが定義されているメモリのオフセット
                        を PUSH する。
    PUSH user_data    ⇒ 転送先のメモリのセグメントセレクタを PUSH する。
    PUSH 0            ⇒ 転送先のメモリのオフセットを PUSH する。
    PUSH 51           ⇒ 転送データのバイト数を PUSH する。
    CALL copy_gate
    JMP start
user_code ENDS
END start,DS:user_data,SS:user_stack

```

図 4・22 転送処理

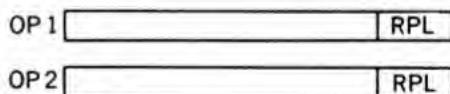
送データのバイト数の順にスタックに PUSH してから、コールゲート COPY\_gate を CALL する。ここで、ソースメモリのセクタ 11000B が OS 内部で定義されたデータセグメント OS\_data のセクタだとしたらどうなるだろうか。このプログラムにおいて、保護のための例外割り込みは発生せず、OS 内部の重要なデータをユーザのメモリ領域にコピーしてしまう。なぜならば、セクタ 11000B を DS に代入

するのは、CPL が 0 で動作する手続き COPY 自身だからである。ユーザプログラムは、11000B をデスティネーションのセクタとして COPY に渡せば、OS のデータを書き換えることもできる。このように、保護された処理の中に、パラ

ARPL OP1,OP2

OP1 = { ワード汎用レジスタ  
ワードメモリ

OP2 = ワード汎用レジスタ



if OP1.RPL < OP2.RPL  
then OP1.RPL ← OP2.RPL, ZF ← 1  
else ZF ← 0

図 4・23 ARPL 命令とその機能

#### 4-8 「トロイの木馬」問題

メータとして送られたデータがシステムに害を及ぼすことを、俗に「トロイの木馬」と呼ぶ。

##### 〔2〕 ARPL 命令による「トロイの木馬」問題の解決

「トロイの木馬」の問題は、セクタを提供するプログラムとそれをセグメントレジスタに代入するプログラムの特権レベルが異なることから発生する。この問題を解決するために、80286 はセクタを提供するプログラムの特権レベル RPL についても検査を行う。また、ARPL 命令を使用して、セクタの RPL を、そのセクタを提供したプログラムの特権レベルに一致させることができる。ARPL 命令は図 4・23

に示すように、オペランドに 2 つのセクタを指定し、もし、OP 1 の下位 2 ビットが OP 2 の下位 2 ビットより数値的に小さければ、OP 1 の下位 2 ビットを OP 2 の下位 2 ビットと同じ値にし、ZF を 1 にする。逆に、OP 1 の下位 2 ビットが OP 2 の下位 2 ビットより数値的に大きい、または等しければ、ZF を 0 にして、OP 1 の値は変わらない。

ARPL 命令は図 4・24 に示すように使用する。CALL 命令が実行された直後、手続きの先頭において BP を PUSH し、SP の値を BP に代入すれば、戻り CS の値は [BP+4]、問題のセクタは [BP+14] で参照される。戻り CS の下位 2 ビットが、セクタ 11000B を提供したプログラムの特権レベルである。したがって、ARPL 命令の左のオペランドに [BP+14]、右のオペランドに [BP+4] の値を指定することによって、セクタの提供者がその特権以上の特権レベルのセクタを送っていないかどうかを検査することができる。このとき、戻り CS の下位 2 ビットは 11B であるから、ARPL 命令によって、[BP+14] の値は 11011B に書き換えられる。この後、DS にセクタ 11011B を代入しようとすれば、 $DPL \geq \text{MAX}(CPL, RPL)$  の規則によって例外割り込みが発生する。また、例外割り込みを発生させるような命令を実行する前に、ZF を調べ

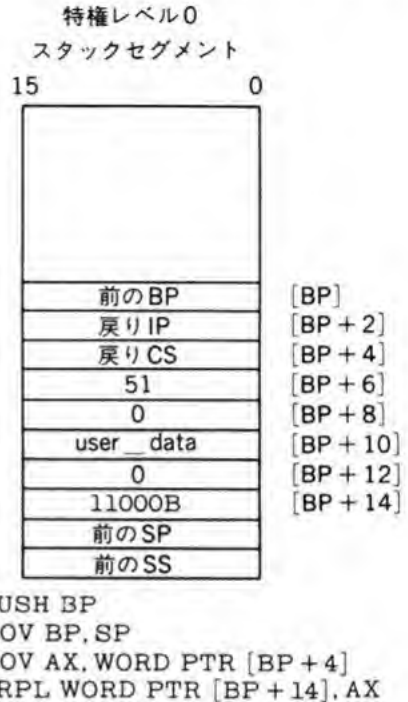


図 4・24 スタックセグメントと ARPL の利用

#### 4 特権レベルによる保護

```
NAME simplified_os

PUBLIC copy

os_data SEGMENT RW
    buff DB 'THIS DATA SHOULD BE PREVENT FROM THE OUTER LEVEL.'

os_data ENDS

os_stack STACKSEG 100H

os_code SEGMENT ER
    ASSUME DS:os_data
    ASSUME SS:os_stack

    message DB 'YOU CANNOT READ MY DATA!!!'

stack_format STRUC
    old_bp      DW ?
    return_ptr  DD ?
    count       DW ?
    destin_ptr  DD ?
    source_ptr  DD ?
stack_format ENDS

copy PROC FAR
    PUSH BP
    MOV BP,SP
    PUSH DS
    PUSH ES
    PUSH SI
    PUSH DI
    PUSH CX

    MOV AX,WORD PTR [BP].return_ptr+2
    ARPL WORD PTR [BP].source_ptr+2,AX
    JZ error

    LDS SI,[BP].source_ptr
    LES DI,[BP].destin_ptr
    MOV CX,[BP].count
    CLD
```

```

    REP MOVSB

    JMP exit

error:
    MOV AX,SEG message
    MOV DS,AX
    MOV SI,OFFSET message
    LES DI,[BP].destin_ptr
    MOV CX,LENGTH message
    CLD
    REP MOVSB

exit:

    POP CX
    POP DI
    POP SI
    POP ES
    POP DS
    POP BP
    RET 10
copy ENDP
os_code ENDS
END

```

図 4・25 「トロイの木馬」問題の解決

て条件 JMP 命令で処理の方向を変えることができる。ARPL 命令を用いて「トロイの木馬」の問題を解決した手続き COPY のプログラムリストを図 4・25 に示す。

## 4 特権レベルによる保護

### 4-9 I/O 参照の保護

80286 は 16 M バイトのメモリ空間とともに、64 K バイトの I/O 空間をもつ。この I/O 空間についても、メモリのセグメントと同様に特権による保護が実行される。I/O 空間の特権レベルは図 4・26 に示すように FLAG の IOPL の 2 ビットに定義され、これはセグメントディスクリプタの DPL に相当する。IN 命令、OUT 命令を初めとして、I/O 空間を参照する命令を実行するときは

$$\text{IOPL} \geq \text{CPL}$$

の条件を満足しなければならない。たとえば、IOPL = 0 であるとき、CPL = 3 の状態で IN 命令を実行すれば、80286 はタイプ 13 の割り込みを発生する。

このように、ユーザプログラムには直接的な I/O 参照を禁止し、実際の I/O 参照を OS レベルの処理で実行するようにシステムを作ることもできる。



I/O 空間を参照する命令において

$$\text{IOPL} \geq \text{CPL}$$

の条件を満足しなければならない。

図 4・26 I/O 空間の特権レベル

## 5. 割り込み処理

プログラムとは独立な原因によって処理の流れを変更することを**割り込み**と呼ぶ。また、割り込みによって実行される手続きを**割り込み手続き**と呼ぶ。割り込みは周辺装置のサービス、CPU 内部または外部で発生する例外処理、デバッグなどに利用される。80286 は一度に 256 種類までの割り込み手続きを定義することができ、それぞれの割り込み手続きを 0 から 255 までのタイプ番号で識別する。リアルモードにおける割り込みは、いくつかの新しい内部割り込みが定義されている以外は、8086 のものと同じである。ここでは、主にプロテクトモードにおける割り込みについて述べる。

## 5-1 割り込みの原因

割り込みが発生する原因は、ハードウェア割り込み、内部割り込み、ソフトウェア割り込みの3種類に分類することができる。ハードウェア割り込みは、80286に外部から割り込み信号を与えることによって、割り込みを発生させるものである。図5・1に示すように、80286は3種類の割り込み信号入力端子をもつ。NMI端子はハードウェアシステムにとって致命的な現象を発見するために使用する。たとえば、メモリにパリティエラーを検査する回路を組み込んでおき、パリティエラーが発生した場合は、NMI端子にトリガ信号を供給するようにする。このとき、80286は無条件にタイプ2の割り込みを発生する。

周辺装置などからの割り込み信号は、INTR端子に入力する。INTR端子への割り込み信号は制御フラグのIFによってマスクすることができる。IFが1であるときに、INTR端子にHighの信号が入力されれば、80286は割り込みアクリッジサイクルと呼ぶ一種のリードバスサイクルを実行し、外部の割り込み制御回路から1バイトの割り込みタイプを入力して、割り込みを発生させる。しかし、IFを0にしておけば、80286はINTR端子の信号を無視する。80286の割り込み制御回路には、8086と同様に8259Aを使用する。

ERROR端子は数値演算プロセッサ80287からの例外割り込みに使用する。

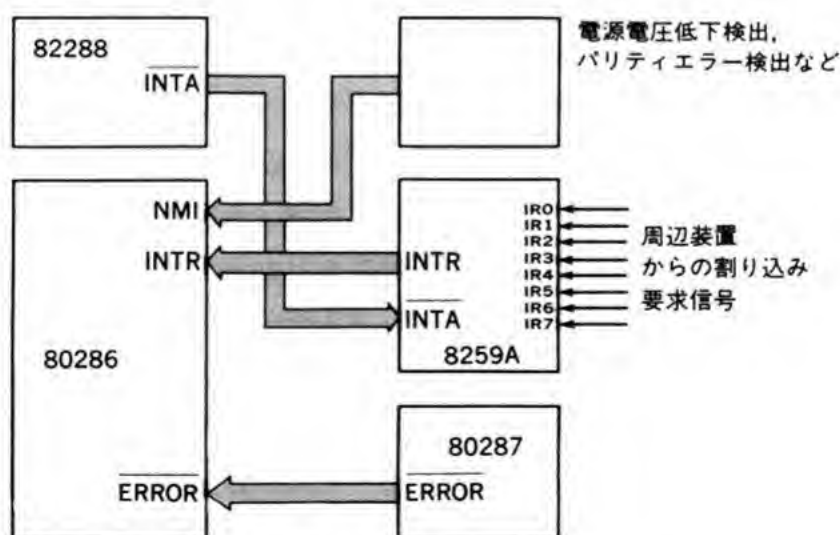


図 5・1 ハードウェア割り込み

## 5-1 割 り 込 め の 原 因

80287 については第 9 章において述べる。

内部割り込みとは、80286 が例外処理の必要を認めたとき、内部で自動的に発生する割り込みであり、表 5・1 に示す割り込みが定義されている。

ソフトウェア割り込みは、INT 命令などの命令によって割り込み処理を実行する。表 5・2 にソフトウェア割り込みの命令を示す。

表 5・1 80286 内部割り込み

割り込みタイプ	割 り 込 め の 原 因
0	DIV, IDIV 命令において、結果の商が AL または AX で表現できない大きさになったとき。
1	TF ビットが 1 のとき、1 命令を実行した後発生する。デバッグのシングルステップに利用できる。
6	不正命令コードの実行。80286 が定義されていない命令コードを実行しようとしたとき発生する。
7	80287 が存在しないとき (EM = 1 かつ MP = 0)、ESC 命令が実行された。または 80287 が存在するとき (EM = 0, MP = 1)、タスクスイッチ直後に (TS = 1) ESC 命令または WAIT 命令が実行された。
8	2 重例外割り込みの場合。
9	80287 オーバランが発生した。すなわち、80287 が扱う 1 ワード以上のデータの一部分がセグメントリミットを越えたときに発生する。
10	不正な TSS が参照された。
11	セグメントが存在しない (P = 0)。
12	スタックセグメントに関する例外割り込みの場合。
13	一般プロテクトエラーが発生した場合の例外割り込みの場合。

表 5・2 ソフトウェア割り込み

割り込み命令	発生する割り込みタイプ
INTn (n = 0 ~ 255)	0 ~ 255
INTO	4
INT 3	3
BOUND	5

### 5-2 割り込みのプロセスと IDT

〔1〕 **割り込み処理と IDT** 5-1 で述べた原因によって割り込みが発生するが、80286 は各命令の実行終了時に割り込みの発生があるかどうかを認識する。しかし、例外として、MOV 命令、POP 命令のように SS に値を代入する命令の後では、外部からのハードウェア割り込みが発生していてもそれを無視する。その次の命令の実行終了時に割り込み処理を実行する。これは、SS だけを変更して、SP を未変更の状態です割り込み処理を実行したとすれば、誤ったメモリ領域をスタックとして使用することになるからである。したがって、SS を変更する命令を実行した直後に、SP を変更する命令が必ず実行できるようになっている。

80286 が割り込み発生を認識してから、割り込み手続きを実行するまでに、図 5-2 に示すような手順を実行する。このように割り込み手続きの最初の命令を実行する前に、自動的に実行される処理を**割り込み処理**と呼ぶことにする。割り込みタイプは、すでに定義されているもの、INT 命令のオペランドで指定するもの、割り込みアクノリッジサイクルを実行して、外部からリードするものがある。割り込みタイプが決まれば、図 5-3 に示す **IDT** (interrupt descriptor table) と呼ばれる特別なテーブルのオフセットが [割り込みタイプ×8] の領域から 6 バイトのデータを読む。IDT は、プロテクトモードにおける割り込みベクトルテーブルであり、IDT のベースアドレスとサイズは IDTR によって定義される。IDT と IDTR の関係は、GDT と GDTR の関係とまったく同じであり、LGDT 命令、SGDT 命令によって、メモリから GDTR に値を代入したり、また、GDTR の値をメモリに書いたりできるように、IDTR についても、LIDT 命令、SIDT 命令を使用して、メモリから IDTR に値を代入したり、IDTR の値をメモリに書いたりすることができる。

〔2〕 **割り込みゲート、トラップゲートによる制御移行** しかし、IDT に定義するディスクリプタは、図 5-4 に示すような**割り込みゲート**または**トラップゲート**の 2 種類のゲートである。これらのゲートは、ワードカウントを指定しないことを除けば、コールゲートとよく似ていて、割り込みによって実行される手続きの実行開始アドレスを定義する。割り込みゲートとトラップゲートの使い分けは、ただ 1 つ、割り込み処理において IF を 0 にするか、しないかだけである。

## 5-2 割り込みのプロセスと IDT

割り込み手続きの実行中に、INTR 端子による割り込みを無視したいときは、IF を 0 にするために割り込みゲートを使用する。逆に、割り込み手続きの実行中でも、INTR 端子からの新たな割り込み要求を受ける場合は、IF の値を変化させないトラップゲートを使用する。

割り込み処理において、FLAG、戻りアドレスのセグメントセクタ、戻りアドレスのオフセットの順に、自動的にスタックに PUSH される。さらに、例外

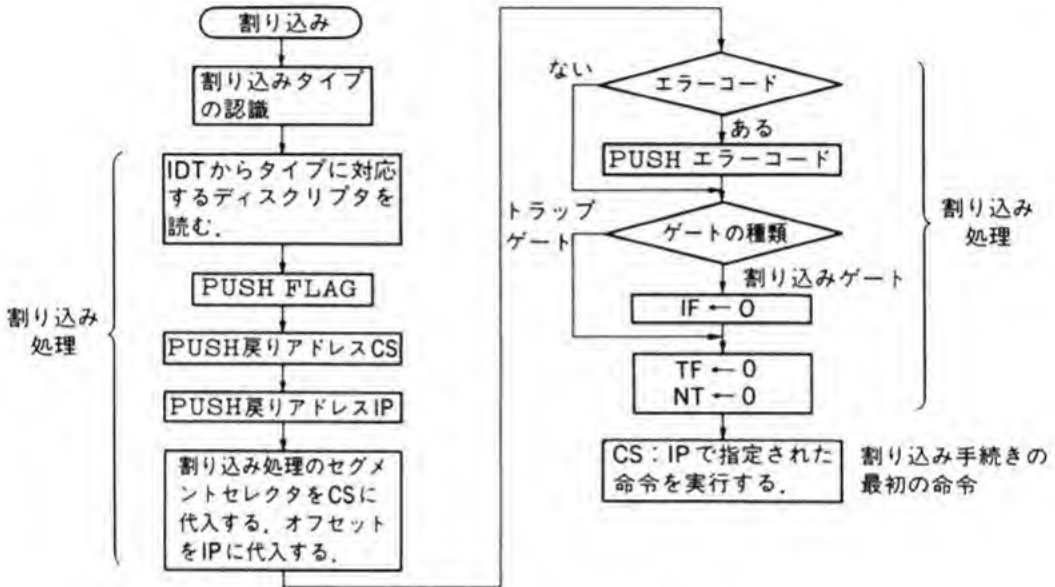


図 5・2 割り込みの手順

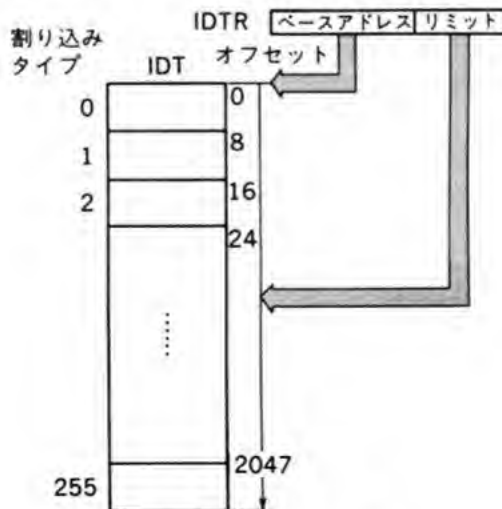
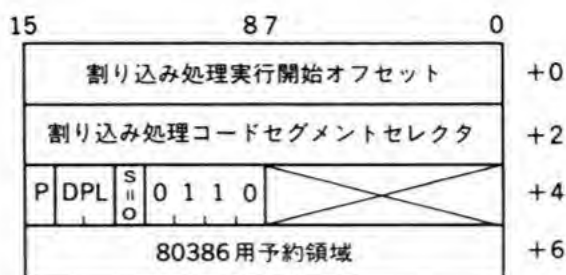
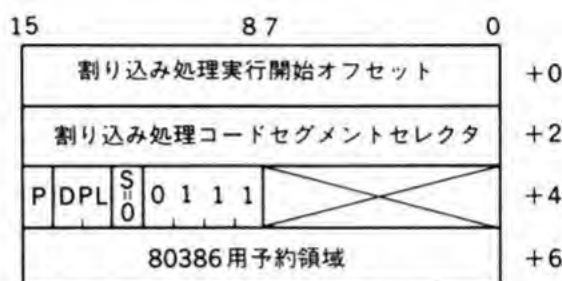


図 5・3 IDT と IDTR

## 5 割り込み処理

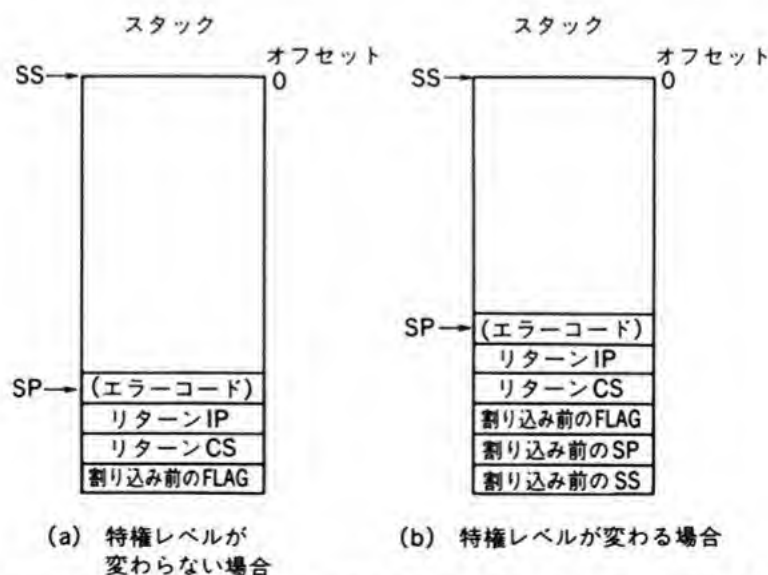


(a) 割り込みゲート



(b) トラップゲート

図 5・4 割り込みゲートとトラップゲート



注) 割り込みの原因によって、エラーコードを PUSH する場合と、しない場合がある。

図 5・5 割り込み発生後のスタック

割り込みの中には、エラーコードを PUSH するものがある。したがって、割り込み発生後のスタックは、図 5・5 に示ようになる。

また、割り込みゲート、トラップゲートを使用して、割り込み手続きに制御を移行する場合も、コールゲートを参照する CALL 命令の場合と同じ規則に従って、特権レベルを変更することができる。特権レベルが変わるときは、やはりスタックセグメントも自動的に更新される。特権レベルが変わったときの新しいスタックの様子を図 5・5(b) に示す。

〔3〕 **割り込み処理の優先** 80286 が割り込みを認識したとき、2 つ以上の割り込みが同時に発生している場合は、表 5・3 に示す順序で処理される。たとえば、NMI 端子へのトリガ入力によって、タイプ 2 の割り込み要求と DIV 命令の実行によって、タイプ 0 の割り込みが同時に発生しているときは、タイプ 0 の割り込み処理を先に実行する。しかし、80286 は割り込み手続きを実行する前に再び、割り込みの発生を認識するので、タイプ 0 の割り込み手続きを実行する前に、タイプ 2 の割り込み処理を実行する。したがって、タイプ 2 の NMI の割り込み手続きの方が先に実行される。

〔4〕 **割り込み手続きの定義** 以上のような割り込み処理を実行した後、初めて割り込み手続きの最初の命令が実行される。

割り込み手続きの定義は CALL 命令で引用される手続きと変わるところはない。ただし、割り込み手続きに制御を移行する割り込みゲートまたはトラップゲートを IDT に定義すればよい。また、割り込み手続きから元のプログラムに制

表 5・3 割り込み処理の順序

割り込みを処理する順序	割り込みの種数
1	命令実行時に内部で発生する例外割り込み
2	TF を 1 に設定したときに発生するシングルステップ
3	NMI
4	タイプ 9 の 80287 オーバーラン
5	INTR

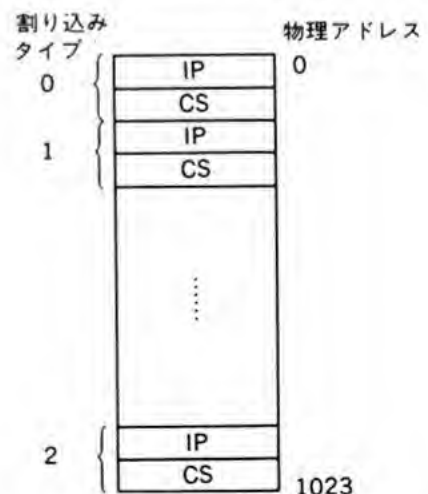


図 5・6 リアルモードにおける IDT

## 5 割 り 込 み 処 理

御を戻すためには IRET 命令を使用する。IRET 命令はスタックから戻りアドレスの IP, CS を順に POP し、最後に FLAG を POP して割り込みの入った元のプログラムに制御を移行する。また、割り込みによる制御移行で特権レベルが変化している場合には、RET 命令と同じようにスタックを切り換えて低い特権のプログラムに戻るができる。

また、80286 は DIV 命令、BOUND 命令など、命令を実行中に発生した例外割り込みについては、割り込みの原因となった命令の先頭アドレスのセグメントとオフセットをスタックに PUSH するから、IRET 命令を使用するときには注意する必要がある。すなわち、割り込みの原因を取り除いてからそのまま IRET 命令を実行して、前に割り込みの原因となった命令を再実行するか、またはスタックに保存されている戻りアドレスの IP を書き換えてから IRET 命令を実行して割り込みの原因となった命令をスキップして、次の命令に戻るようにする。

〔5〕 **リアルモードの IDT**    なお、リアルモードでの IDT は 8086 の割り込みベクトルテーブルとまったく同じ状態になっている。すなわち、図 5・6 に示すように物理アドレスの 0 番地から 1023 番地までに、割り込みタイプ 0 から 255 までのそれぞれの割り込み手続きの実行開始アドレスだけが定義されたものである。リアルモードにおいても、LIDT 命令を使用し、IDTR の値を変更することによって、IDT の定義アドレスを変更することができるが、これはリアルモードからプロテクトモードに移行するときの初期設定として実行するべきものであり、リアルモードで動作するシステムにおいては絶対に使用してはならない。

## 5-3 ハードウェア割り込み

ハードウェア割り込みと呼ばれるものには、5・1で述べたように NMI, INTR,  $\overline{\text{ERROR}}$  の各端子から入力される割り込み信号が原因となって発生するものがある。ここでは、NMI, INTR 端子に入力される信号の形式について述べる。

NMI はメモリパリティエラー、電源電圧低下など、システムにとって致命的な状況に対する例外処理を実行するために使用される。NMI 端子には図 5・7 に示すような、4 クロック以上の間 Low を保ち、High に立ち上がった後、4 クロック以上の間、High を保持する信号を入力する。NMI 信号は 80286 内部でラッチされ、タイプ 2 の割り込みを発生する。NMI の割り込み処理を実行中に、再び NMI 割り込み信号が入力されると、その信号は 80286 内部でラッチされるが、IRET 命令を実行するまで処理されない。IRET 命令を実行した後に、初めてラッチされていた NMI 割り込みが処理される。

INTR 端子はレベルセンスの端子であり、各命令の終了時に INTR 端子の電圧レベルを検査し、High であれば、割り込みアクノリッジサイクルと呼ぶ一種のリードバスサイクルを実行し、割り込み制御回路 8259 A から 1 バイトの割り込みタイプをリードする。INTR 端子からの割り込み要求は IF によってマスクすることができる。

80286 が実行する割り込みアクノリッジサイクルを図 5・8 に示す。

また、80286 と 8259 A の接続の様子を図 5・9 に示す。8259 A の使用は基本的には 8086 の場合と同じであるが、図 5・8 に示すように、割り込みアクノリッジサイクルのサイクル 1、サイクル 2 それぞれに少なくとも 1 つのウェイトサイクルを入れなければならないことに注意する。このウェイトサイクルは一般のバスサイクルと同様に  $\overline{\text{READY}}$  によって制御する。

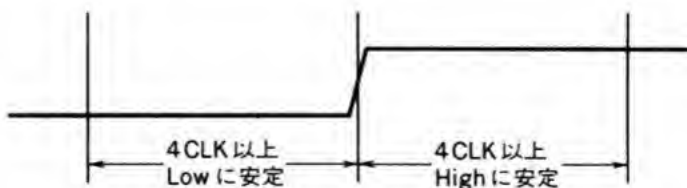


図 5・7 NMI 割り込み信号

## 5 割り込み処理

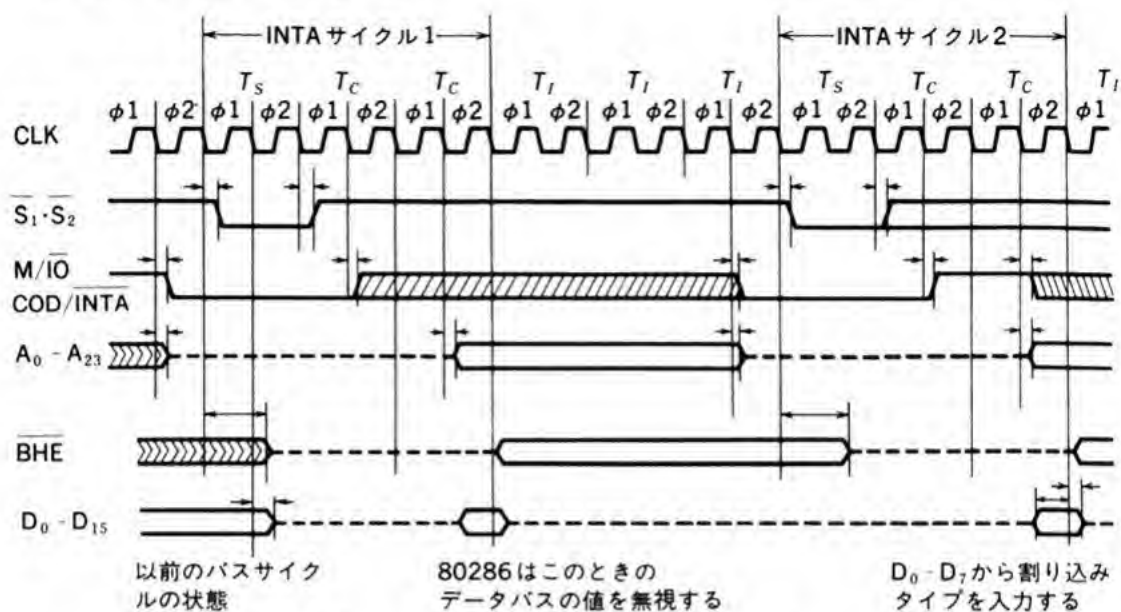


図 5・8 80286 割り込みアクノリッジサイクル

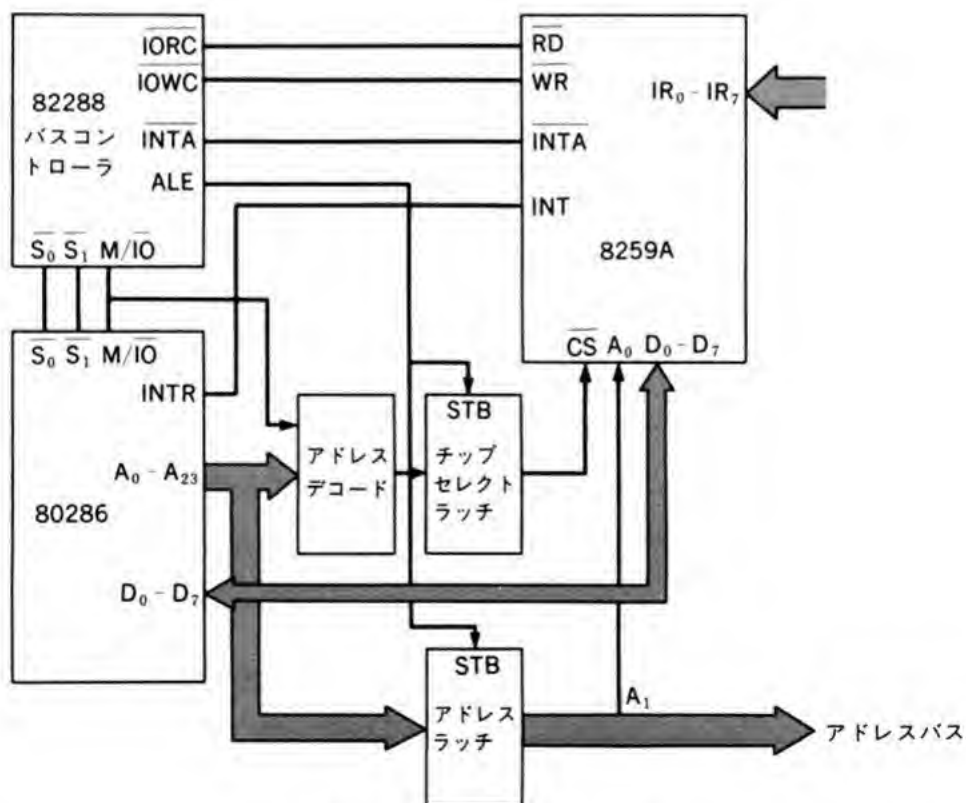


図 5・9 80286 と 8259A の接続

## 5-4 ソフトウェア割り込み

表5・2に示したソフトウェア割り込みは、INT 命令などのソフトウェア命令によって割り込み処理を実行するもので、CALL 命令によって手続きに制御を移行させるのとはよく似ている。ソフトウェア割り込みがCALL 命令と異なるところは、スタックにFLAG、戻りアドレスのCSとIPをPUSHして、IDTに定義された割り込みゲートまたはトラップゲートを介して、制御を移行させるこ

```

CODE1 SEGMENT ER

    OVERFLOW_MSG DB "*** OVER FLOW ERROR **",ODH,OA

    INT4 PROC FAR ←
        PUSH DS
        PUSH ES
        PUSH AX

        PUSH SEG OVERFLOW_MSG
        PUSH OFFSET OVERFLOW_MSG
        PUSH LENGTH OVERFLOW_MSG
        CALL DISPLAY
        MOV I1,MAX_VALUE

        POPA
        POP ES
        POP DS

        IRET
    INT4 ENDP
CODE1 ENDS

CODE2 SEGMENT EO
    :
    :
    ADD I1,5
    INTO
    :
CODE2 ENDS

```

もしオーバーフローが発生すれば(OF=1),  
タイプ4の割り込みが発生する。

図 5・10 INTO 命令の利用

## 5 割 り 込 み 処 理

とである。したがって、INT 命令はシステム内部のどの応用プログラムからでも引用されるような最も基本的な手続を引用するときに便利である。

〔1〕 **INTO 命令** INTO 命令は INT 命令の中で特別な命令である。これは OF が 1 のとき、タイプ 4 の割り込みを発生するが、OF が 0 のときは、そのまま次の命令を実行する。INTEGER タイプのデータの演算の後に INTO 命令を書くことによって、整数演算でオーバフローが発生したときの例外処理をタイプ 4 の割り込みによって実行することができる。図 5・10 に、整数変数 11 に 5 を加算した後、オーバフローが発生すれば、タイプ 4 の割り込み処理によって、"**\*\*OVER FLOW ERROR\*\***" のメッセージを CRT に表示するプログラム例を示す。もちろん、このとき IDT のオフセット 32 から 8 バイトの領域に、手続き INT 4 に制御を移行するための割り込みゲートまたはトラップゲートを定義しておかなければならない。

〔2〕 **プログラムデバッグに利用する割り込み** INT n 命令は 2 バイトの命令であるが、INT 3 だけは 1 バイトの命令である。これは、プログラムのデバッグにおいて、実行を中断するブレイクポイントを指定するために使用する。すなわち、プログラムで中断したい部分の命令を INT 3 命令に置き換えておいて、プログラムを実行すれば中断したいところでタイプ 3 の割り込みに入る。タイプ 3 の割り込みが発生したとき、INT 3 命令を元の命令で、再び置き換えておけばよい。

プログラムのデバッグにおいて、1 命令ずつ実行したい場合は、制御フラグの TF を利用することができる。TF を 1 にセットしておけば、80286 は 1 命令を実行した後、タイプ 1 の割り込みを発生する。

ここで、TF だけを 1 にする命令はないので、実際の処理では図 5・11 に示すように、スタックに FLAG、実行する命令の CS、IP を PUSH した後で IRET を実行することによって、FLAG を書き換えて目的の命令を実行することができる。もちろん、IRET 命令を実行する前に、スタックに入っている FLAG の TF を 1 にしておく。この処理の流れを図 5・12 に示す。

〔3〕 **BOUND 命令** BOUND 命令は配列のレンジチェックを行う命令で、図 5・13 に示すように使用する。配列を定義するとき、配列の最小と最大のオフセットを 4 バイトの変数 BUFRANGE に定義しておけば

BOUND BX, BUFRANGE

## 5-4 ソフトウェア割り込み

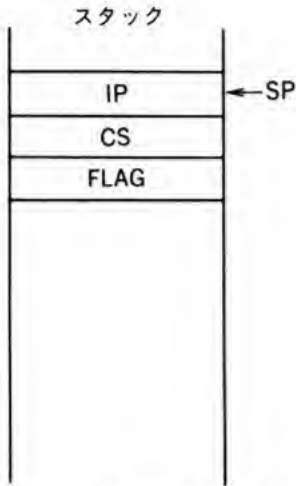


図 5・11 IRET 命令を実行する前のスタック

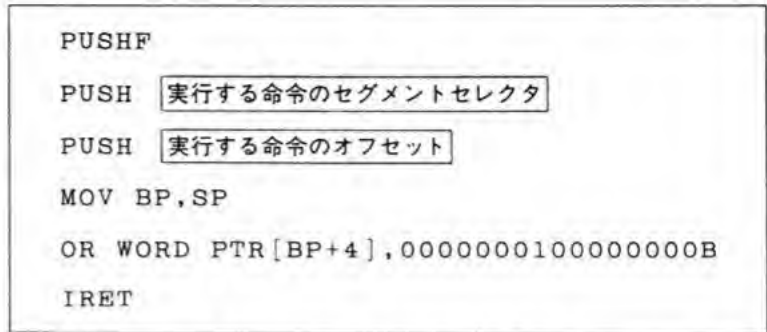
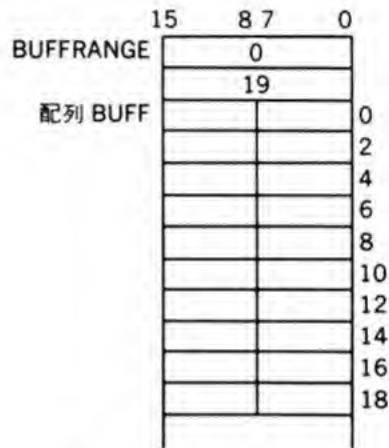


図 5・12 1 命令だけを実行する方法



```

BOUND BX, BUFRANGE
MOV BUFF [BX], 0FFH
    
```

図 5・13 BOUND 命令の利用

のような命令によって、BX の値が BUFRANGE の下位ワードに定義された値より大きく、BUFRANGE の上位ワードに定義された値より小さいかどうかを検査できる。図 5・13 の場合、 $BX \geq 0$ 、かつ、 $BX \leq 19$  のとき、そのまま次の命令を実行するが、BX の値が 0 から 19 の範囲に入っていないとき、タイプ 5 の割り込みを発生する。ここで、BOUND 命令の左のオペランドは、BX 以外にワード汎用レジスタであればなにを指定してもよい。

## 5-5 内部割り込み

命令の実行中に 80286 が自動的に発生する割り込みを内部割り込みと呼ぶ。内部割り込みの中で、保護例外と呼ばれるものについては第 7 章において述べる。ここでは、表 5・4 に示すような保護例外以外の内部割り込みについて述べる。

DIV 命令、IDIV 命令を実行したとき、商はバイト操作のときは AL、ワード操作のときは AX に残るが、0 で割り算を実行した場合とか、商が AL、AX で表現できる値より大きい場合には、タイプ 0 の割り込みを発生する。

タイプ 1 の割り込みについては、5-4 で述べたとおりである。

タイプ 6 の割り込みは、80286 が定義されていない不正な命令コードを実行したときに発生する。

80286 の内部割り込みの戻りアドレスは、タイプ 1 のシングルスステップ以外は割り込みの原因となった命令の先頭アドレスがスタックに PUSH されることに注意する。たとえば、8086 では、タイプ 0 の割り込みのとき、DIV 命令または IDIV 命令の次の命令のアドレスが、戻りアドレスとしてスタックに PUSH されたのに対して、80286 では、DIV 命令または IDIV 命令自身の先頭アドレスが、戻りアドレスとしてスタックに PUSH される。したがって、そのまま IRET 命令を実行すれば、割り込み発生の原因となった命令を再実行する。このことはタイプ 6 の不正命令コード実行の例外割り込みについても同じである。

表 5・4 内部割り込み

割り込み タイプ	原 因	戻りアドレス
0	DIV, IDIV 命令実行時に結果の商が、AL, AX で表現できない。	割り込みの原因となった命令の先頭アドレス
1	TF = 1 となっている場合、1 命令実行後発生する。	次の命令のアドレス
6	80286 で定義されていない、不正命令コードを実行しようとした。	割り込みの原因となった命令の先頭アドレス

## 6. タスクとタスクスイッチ

80286 のプロテクトモードでの特徴は、厳密なメモリ管理とタスク管理である。8 MHz で動作する 80286 では、約  $23\mu\text{s}$  でタスクスイッチを実行することができる。第 6 章では、タスクとは何か、80286 はタスクをどのように管理するのか、そして、タスクスイッチの命令について述べる。

## 6-1 シングルタスク

シングルタスクシステムは図 6・1 に示すように、1つのプログラムがその実行開始から終了まで CPU を専有するシステムである。たとえば、MS-DOS (V3.X までのもの) とか CP/M-86 のような OS を考えればよい。プログラム A を実行するとき、キーボードから入力されたコマンドによって、OS がディスクファイルからプログラムをメモリにロードし、レジスタに必要なデータを初期設定してから、制御をプログラム A に移行する。この後、次のプログラムを実行するためには、プログラム A が実行を終了するまで待たなければならない。

図 6・1 において、プログラム A が終了すると制御はプログラム B に移行する。ここで、プログラム B は OS のコマンド入力処理と考えればよい。OS は再び CRT にプロンプトマークを表示し、次のコマンドが入力されるのを待つ。

このようなシステムでは、一度に一人のユーザしかコンピュータを利用することができない。また、このようなシステムを機械制御に利用することを考えると、必要に応じて要求されたプログラムを十分なレスポンスで実行させることができない。

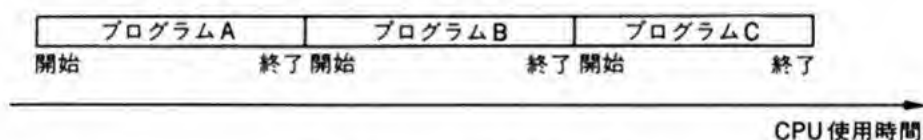


図 6・1 シングルタスクシステム

## 6-2 マルチタスク

〔1〕 **マルチタスクの動作** マルチタスクとは2つ以上のプログラムを同時に実行でき、前のプログラムが実行を終了していなくても、次のプログラムの実行を開始できるシステムである。マルチタスクシステムの応用としては、**タイムシェアリングシステム**、**リアルタイムシステム**がある。もちろん、80286 が本当の意味で、同時に2つ以上のプログラムを実行することはできない。限られた特定の時間を考えれば、80286 は1つのプログラムを実行するだけである。ここで、マルチタスクシステムで実行されるプログラムのことを**タスク**と呼ぶ。プログラムとタスクとは1対1に対応しない。なぜならば、メモリにロードされた1つのプログラムが、2つの別々のタスクとして実行されることがあるからである。

図6・2に3つのタスク A, B, C が実行される様子を示す。80286 の実行時間は分割され、各タスクに分け与えられる。最初、80286 はタスク A を実行するが、時間  $t_1$  から  $t_3$  の間は別のタスクを実行する。したがって、時間  $t_1$  でタスク A の処理を中断してから、時間  $t_3$  で再開するまでの間、レジスタの値などの80286 の状態をメモリ上の特別なセグメントに保存する必要がある。この特別なセグメントを **TSS (task status segment)** と呼ぶ。

TSS はタスクごとに1つずつ定義する。時間  $t_1$  において、80286 の使用権をタスク A からタスク B に渡すとき、現在のレジスタの状態をタスク A の TSS に保存してからタスク B の TSS に保存されているレジスタの状態を80286 のレジスタに代入する。この後、80286 が以前と同様に実行を続ければタスク A の実行が完全に終了していないにもかかわらず、80286 はタスク A とは関係のない別のタスクを実行することができる。

このことは、時間  $t_2$  においてタスク B からタスク C に切り換わる時も同様である。時間  $t_3$  において、再びタスク A に80286 の使用権が渡るとき、80286 のレジスタの状態がタスク C の TSS に保存されてからタスク A の TSS から80286 のレジスタに、時間  $t_1$  において保存された状態が再び代入される。

もし、図6・2に示すシステムにおける各タスクが、3人のユーザがそれぞれ3台のターミナルを使用して、別々に実行しているプログラムであるならば、それぞれのユーザは1つの80286を他のユーザと分け合っていることには気づかな

## 6 タスクとタスクスイッチ

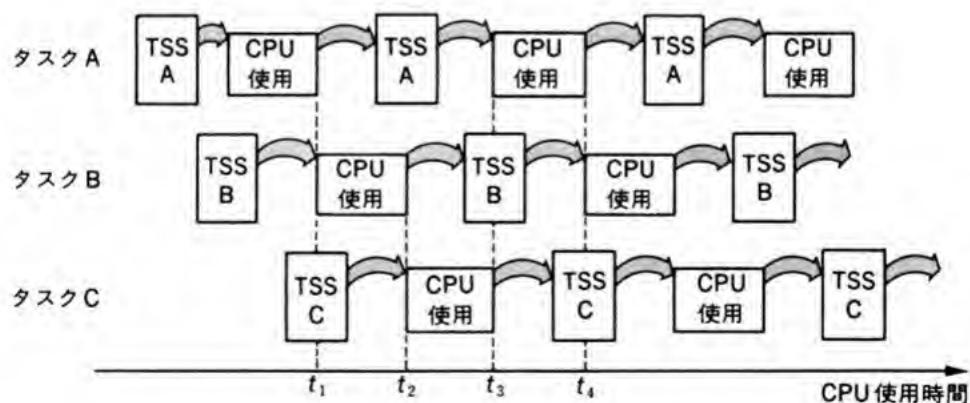
いだろう。

また、マルチタスクシステムを機械制御に利用するならば、新しい仕事の要求に対して、実時間（リアルタイム）で応答できるシステムを作ることができる。前者の例では、タイマを使用して一定時間ごとに必ずタスクを切り換えるようにする。このようなシステムを**タイムシェアリングシステム**と呼び、実行可能なタスクは一定時間ごとに必ず CPU のサービスを受ける。

また、後者のリアルタイムシステムでは、新しいデータが入力されるなどの何らかの事象の発生によってタスクを切り換える。もちろん、このようなシステムでは、実行可能なタスクの中で次にどのタスクに CPU を与えるかを決定し、管理する**タスク管理**のプログラムが必要である。

〔2〕 80286 の TSS TSS は一種のセグメントであるから、最大 64 K バイトの大きさまで定義することができるが、80286 ではそのオフセット 0 から 43 までの構造が図 6・3 に示すように決められている。また、TSS は図 6・4 に示す **TSS ディスクリプタ** によって定義され、TSS ディスクリプタのセレクトアによって TSS を一意に識別することができる。なお、TSS ディスクリプタは必ず GDT に定義しなければならない。

TSS ディスクリプタのベースアドレス、リミットには、TSS が定義されるベースアドレスと TSS のリミットを指定する。ここで、TSS のリミットは必ず 43 以上の値でなければならない。TSS ディスクリプタのアクセスライトの P、DPL も他のディスクリプタのものと同様である。



ここで TSS (= Task Status Segment) は 80286 のレジスタの値などの内部状態を格納する特別なセグメント。

図 6・2 マルチタスクシステム

## 6-2 マルチタスク

15	0	オフセット
バックリンクセクタ	0	
特権レベル0 SP	2	
特権レベル0 SS	4	
特権レベル1 SP	6	
特権レベル1 SS	8	
特権レベル2 SP	10	
特権レベル2 SS	12	
IP (実行開始オフセット)	14	
FLAG	16	
AX	18	
CX	20	
DX	22	
BX	24	
SP	26	
BP	28	
SI	30	
DI	32	
ES	34	
CS	36	
SS	38	
DS	40	
LDTセクタ	42	

図 6・3 TSS  
(Task Status Segment)

TSS ディスクリプタを参照するときは (TSS ディスクリプタを参照する命令は後に述べるように CALL 命令, JMP 命令などである), P=1 でなければならないし, また DPL に関して,  $DPL \geq$

MAX (CPL, RPL) でなければならない. TSS ディスクリプタの B は 80286 が自動的に 1 にセットしたり, また 0 にクリアするビットで, B=1 のとき TSS ディスクリプタの指定するタスクが処理中であることを表す.

TSS のオフセット 44 からの領域は, たとえば数値演算プロセッサ 80287 のレジスタ状態を保存するなど, 必要に応じてユーザが任意に使用すればよい. TSS のオフセット 14 から 41 までの 14 ワードに, 80286 のすべてのレジスタの状態が保存される. TSS のオフセット 2 から 13 までは, 特権レベルが変わったときに使用される SS, SP の初期値を保存する. ここで, より低い特権レベルから特権レベル 3 に移行することはありえないから, 特権レベル 3 の SS, SP の初期値を



図 6・4 TSS ディスクリプタ

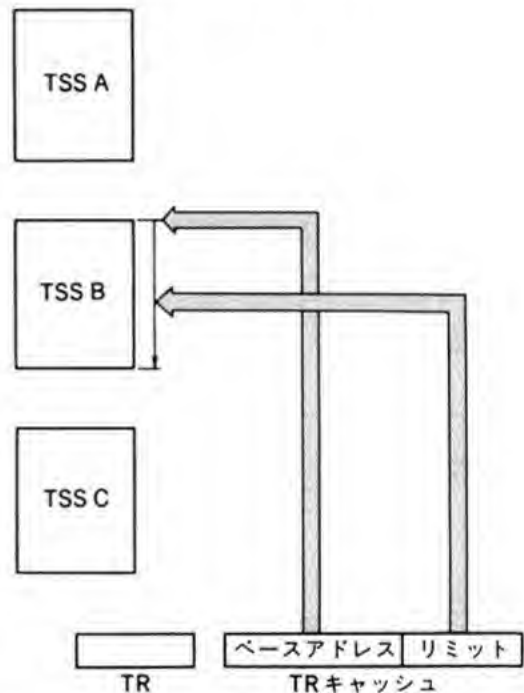


図 6・5 TR と TR キャッシュ

## 6 タスクとタスクスイッチ

記録する必要はない。オフセット 42 から始まる 1 ワードには、そのタスクで使用する LDT の LDT ディスクリプタのセクタを記録する。これによって、タスクごとに別々の LDT を指定することができる。最後に、オフセット 0 から始まる 1 ワードは、タスクのバックワードリンクで、以前に実行されていたタスクの TSS ディスクリプタのセクタを記録するために使用する。

〔3〕 **TSS の定義** TSS は図 6・5 に示すように、タスクごとに 1 つずつ定義する。3 つのタスク A、B、C を実行するとき、3 つの TSS を定義し、さらに 3 つの TSS ディスクリプタを GDT に定義する。現在実行中のタスクの TSS は、TR と TR キャッシュによって定義される。TR と TR キャッシュの扱いは、セグメントレジスタとセグメントキャッシュの場合とよく似ている。図 6・6 に示す LTR 命令を使用して、TSS ディスクリプタのセクタを TR に代入するとき、TSS ディスクリプタのリミットとベースアドレスが自動的に TR キャッシュに代入される。このとき、GDT に定義されている TSS ディスクリプタの B が自動的に 1 になる。また、TR の値は STR 命令によって読むこともできる。

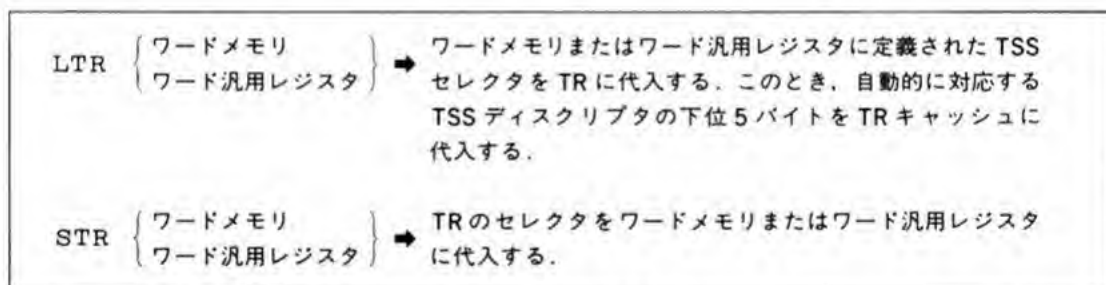


図 6・6 TRに関連する命令

## 6-3 LDT と LDT ディスクリプタ

マルチタスクシステムにおいて、タスク間で共有するディスクリプタは GDT に定義し、特定のタスクだけで参照するディスクリプタは LDT に定義する。したがって、LDT はタスクごとに1つずつ定義し、タスクと LDT の対応を決めるために、図 6・7 に示すように、TSS のオフセット 42 から始まる1ワードに LDT ディスクリプタのセクタを定義する。図 6・7 は3つのタスクの中でタスク B が実行されるとき、TR、TR キャッシュ、LDTR、LDTR キャッシュの

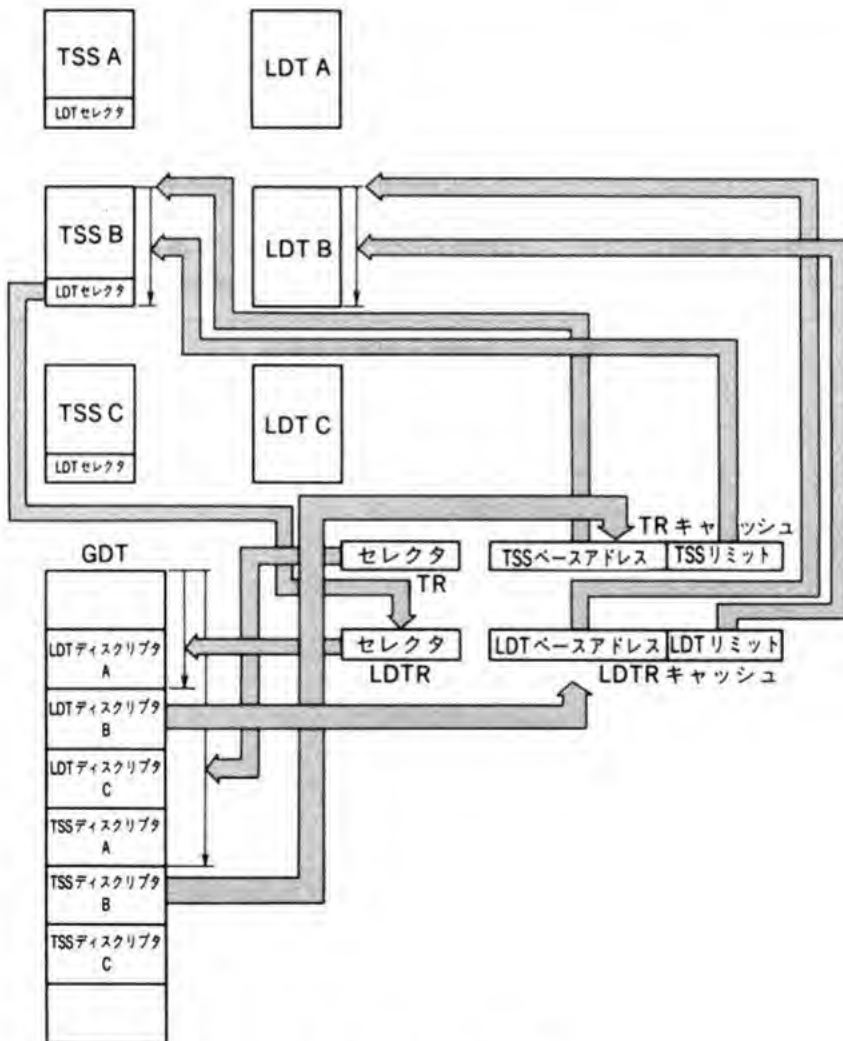


図 6・7 TSS と LDT

## 6 タスクとタスクスイッチ

状態を示す。

現在実行中のタスクで使用される LDT は、TSS を指定する方法と同様に、LDTR、LDTR キャッシュによって定義される。LDTR、LDTR キャッシュの扱いは、TR、TR キャッシュの場合とまったく同様である。図 6・8 に示す LLDT 命令によって、LDTR に LDT ディスクリプタのセクタを代入するとき、自動的に LDTR のセクタで指定される LDT ディスクリプタのリミットとベースアドレスが LDTR キャッシュに代入される。

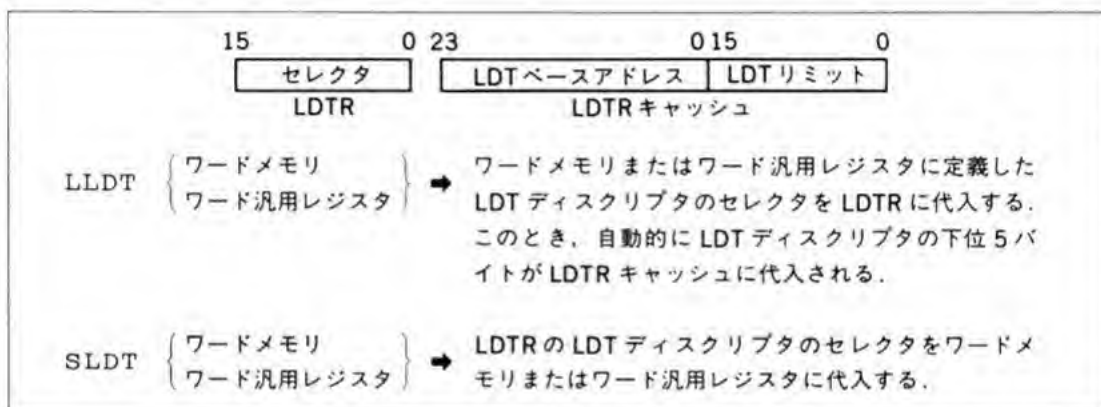


図 6・8 LDTR と関連する命令

## 6-4 タスクの定義

80286において、タスクを実行するまでには、図6・9に示すように GDT, LDT, TSS をメモリに定義し、GDTR, LDTR, TR に必要な値を定義しなければならない。ただし、LDTR と TR は、セグメントレジスタと同じようにキャッシュをもち、GDT からキャッシュに自動的にディスクリプタが代入されるから、LDT ディスクリプタ、TSS ディスクリプタは必ず GDT に定義しておく必要がある。

GDT の定義から、1つのタスクを実行するまでの手順を以下に示す。

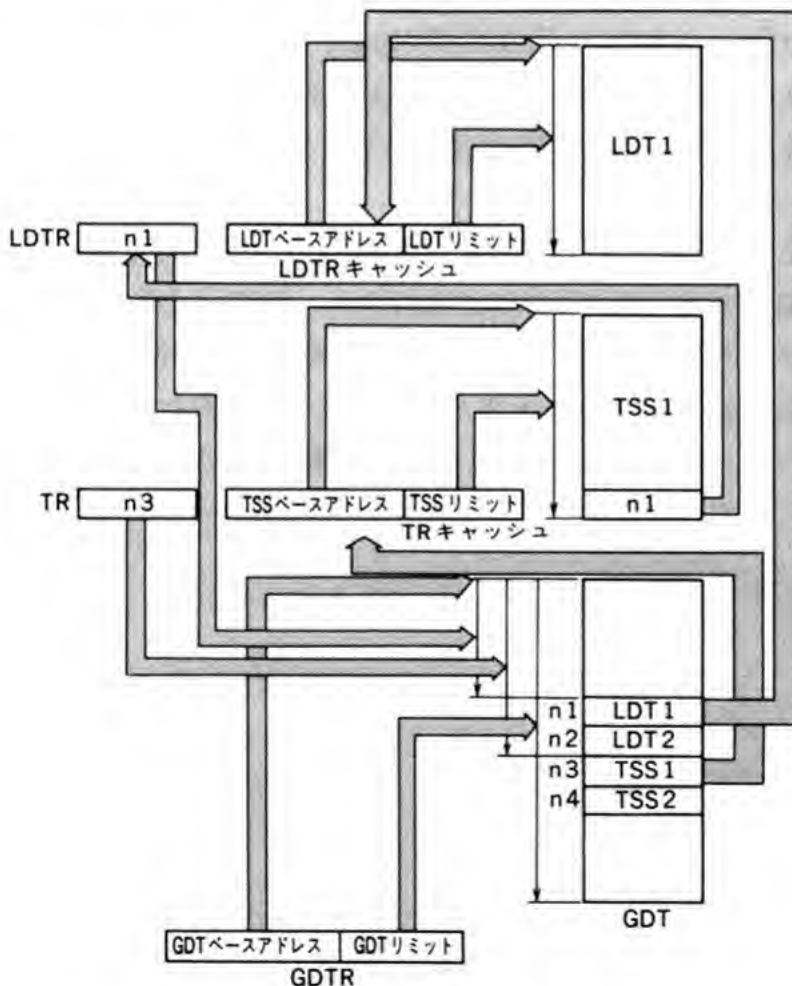


図 6・9 タスクの定義

## 6 タスクとタスクスイッチ

(1) メモリに GDT を書く．GDT には必要なセグメントディスクリプタ，ゲート，LDT ディスクリプタ，TSS ディスクリプタを定義する．

(2) GDTR に GDT のベースアドレスとリミットを代入し，GDT を使用可能な状態にする．

(3) メモリに IDT を定義する．IDT には割り込みゲートまたはトラップゲートを必要に応じて定義する．

(4) IDTR に IDT のベースアドレスとリミットを代入し，IDT を使用可能な状態にする．

(5) リアルモードからプロテクトモードに切り換える．

(6) メモリにいくつかの TSS を定義する．

(7) メモリにいくつかの LDT を定義する．

(8) LDTR に LDT ディスクリプタのセクタを代入する．

(9) TR に TSS ディスクリプタのセクタを代入する．

(10) TR で指定された TSS に記録されているレジスタの状態を各レジスタに代入する．

(11) TR で指定された TSS に記録されている CS, IP に制御を移行する．

以上で TR で指定されたタスクが実行され，GDT, LDT で定義されたセグメントディスクリプタによって，メモリ管理，特権保護が実行される．また，GDT, LDT に定義されたコールゲートまたは IDT に定義された割り込みゲート，トラップゲートによって制御移行における特権保護が実現される．

## 6-5 タスクスイッチ

〔1〕 **タスクスイッチの動作** 80286 の実行をあるタスクから別のタスクに切り換えることを**タスクスイッチ**と呼ぶ。タスクスイッチは次のような手順によって実行される。

- (1) 現在のレジスタの値を TR で指定される現在の TSS に保存する。
- (2) 次のタスクの TSS ディスクリプタのセクタを TR に代入する。これによって、同時に TR キャッシュに TSS ディスクリプタの下位 5 バイトが代入される。
- (3) 新しい TSS に定義された CS, DS, ES, SS 以外のすべてのレジスタの値を対応するレジスタに代入する。
- (4) TSS のオフセット 42 から始まる 1 ワードに記録されている LDT ディスクリプタのセクタを LDTR に代入する。このとき、同時に LDT ディスクリプタの下位 5 バイトが LDTR キャッシュに代入される。
- (5) 新しい TSS に定義された SS, CS, DS, ES のセクタ値を対応するセグメントレジスタに代入する。

〔2〕 **タスクスイッチ命令** 以上で 80286 のレジスタ状態は、今まで実行されてきたタスクとまったく関連のない新しいタスクに移る。そして、80286 は上述のタスクスイッチを 1 つの命令によって、または割り込みによって実行することができる。8MHz の 80286 では、このタスクスイッチを約  $23\mu\text{s}$  で実行できる。タスクスイッチは図 6-10 に示す命令または割り込みによって発生する。割り込みおよび INT 命令によって発生するタスクスイッチについては 6-6 において述べる。ここではソフトウェア命令によって発生するタスクスイッチについて述べる。

タスクスイッチを実行する命令といっても、特別な命令が追加されているわけではない。far JMP 命令、far CALL 命令のオペランドのセクタが、セグメントディスクリプタとかコールゲートではなく、TSS ディスクリプタを指定するとき、一般の JMP 命令、CALL 命令が実行されるのではなく、タスクスイッチを実行する。また、IRET 命令は FLAG の NT が 0 のときは、第 5 章で述べたように割り込みからのリターン処理を実行するが、FLAG の NT が 1 のとき、IRET 命令はタスクスイッチを実行する。IRET 命令によるタスクスイッチで



図 6・10 タスクスイッチの発生

は、現在のタスクの TSS のオフセット 0 から始まる 1 ワードに定義されているバックリンクセクタが、新しいタスクの TSS ディスクリプタのセクタとなる。このバックリンクセクタには、CALL 命令または割り込みによってタスクスイッチしたとき、自動的に以前のタスクの TSS ディスクリプタのセクタが記録されるようになっている。したがって、CALL 命令によってタスクスイッチした後で、IRET 命令によってタスクスイッチすれば、以前のタスクに帰ることができる。

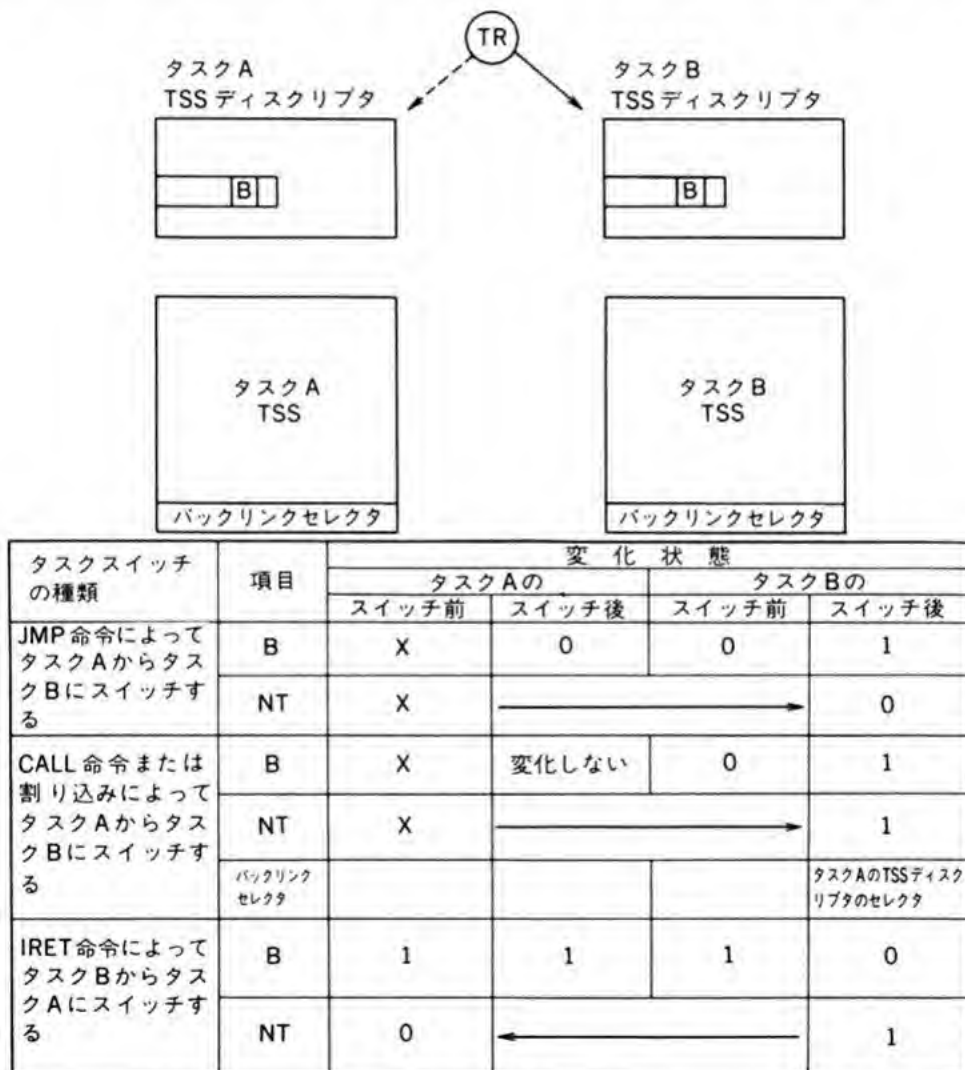
〔3〕 **タスクスイッチ命令の使い分け** JMP 命令によって実行されても、また CALL 命令、IRET 命令によって実行されても、タスクスイッチの動作自体は変わらない。しかし、命令によって少し異なる部分もある。このことを図 6・11 に示す。これらの命令の違いを考慮して、タスクスイッチの命令を状況に応じて使い分ければよい。図では、タスク A からタスク B にスイッチされ、再びタスク B からタスク A にスイッチされる状況を表している。

JMP 命令によって、タスク A からタスク B にスイッチするとき、タスク A の TSS ディスクリプタの B の状態は、スイッチ前は 0 でも 1 でもかまわないが、スイッチ後は自動的に 0 となる。また、タスク B の TSS ディスクリプタの B の状態は、スイッチ前には必ず 0 でなければならない。そして、スイッチ後は自動的に 1 になる。FLAG の NT もタスクスイッチによって変化し、JMP 命令によるタスクスイッチの場合、0 にクリアされる。JMP 命令によるタスクスイッチ

## 6-5 タスクスイッチ

は最も一般に使用されるものであり、図 6・11 の状況においては、タスク A を終了して ( $B \leftarrow 0$ )、タスク B を開始する ( $B \leftarrow 1$ ) ことを表す。したがって、タスク B の処理において、タスク A にスイッチする JMP 命令を実行する場合も何の問題もなく、タスク A に移ることができる。

CALL 命令によるタスクスイッチの場合、B、NT の変化が JMP 命令で実行される場合とは異なる。CALL 命令の場合、タスク A の TSS ディスクリプタの B の値は変化せず、タスク B の TSS ディスクリプタの B が 0 から 1 になる。そして、FLAG の NT が 1 になる。これは、タスク A の処理は、まだ実行中の



ここで、X は 0 でも 1 でも、どちらでもよいことを表す

図 6・11 タスクスイッチの種類

## 6 タスクとタスクスイッチ

まま (B の値が変化しない), 新しいタスク B が起動された ( $B \leftarrow 1$ ) ことを表す. そして, NT が 1 になるのは, タスク B はタスク A の子プロセスであることを表す. また, タスク B の TSS のバックリンクセクタには, その親であるタスク A の TSS ディスクリプタのセクタが記録される. このように, CALL 命令によるタスクスイッチは, 親プロセスが子プロセスを起動するというように考えることができる.

CALL 命令によってタスク A からタスク B にスイッチしたとき, タスク B の処理の中で IRET 命令を実行すると, CALL 命令によるタスクスイッチによって NT=1 となっているから, バックリンクセクタに定義されたタスク A に戻ることができる. IRET 命令によるタスクスイッチは, 子プロセスを終了して親プロセスに戻るというイメージで利用することができる.

### タスクとプロセス

OS の用語として使用されるタスクという言葉を一言で説明することは困難である. 仕事とか処理などの日本語に翻訳してもタスクの意味を説明することにはならない. したがってタスクとは何かと問われたとき, タスクは実行可能なプログラムであると言わざるをえない. 実行可能とはプログラムが CPU, メモリなどのコンピュータ資源を利用可能であることを表す. では, 1 つのタスクは必ず 1 つのプログラムであるかというそうではない. タスクの意味を考えるためにはどうしてもマルチタスクシステムを考えなければならない.

マルチタスクシステムは, メモリに配置された実行可能な複数のプログラムが CPU, メモリなどのコンピュータ資源を分け合いながら処理されるシステムである. ここでプログラムが 1 つのタスクとして処理されるが, もし同種類の 2 つのタスクを実行する場合には, メモリには 1 つだけのプログラムを配置し, それを 2 つのタスクとして 2 重に使用してもよい. それぞれのタスクが扱う入力データは別々のものであり, また出力するデータも異なる. タスクとはマルチタスクシステムにおいて管理される 1 処理単位である.

プロセスという言葉も, 使用する人によって, さまざまな状況においてさまざまな意味で使われている. しかし, 少なくとも UNIX のマニュアルの中で使用されているプロセスの意味は 80286 のタスクと同じである.

## 6-6 タスクゲート

ゲートと呼ばれるディスクリプタには、コールゲート、割り込みゲート、トラップゲートの他に図6・12に示す**タスクゲート**と呼ばれるゲートがある。タスクゲートには、タスクスイッチをするときの新しいタスクの TSS ディスクリプタのセクタを定義する。タスクゲート自体は、GDT, LDT, IDT に定義することができる。タスクゲートを GDT または LDT に定義したとき、far JMP 命令、far CALL 命令のセクタ部にタスクゲートのセクタを指定すれば、タスクゲートに定義したセクタが指定する TSS ディスクリプタのタスクにスイッチできる。このとき、TSS ディスクリプタの B, FLAG の NT, バックリンクセクタの変化は、タスクゲートを参照する命令によって図6・11に示したとおりである。

JMP 命令、CALL 命令がタスクゲートを参照するとき、タスクゲートの P, DPL についてコールゲートの参照の場合と同様に、 $P=1$ ,  $DPL \leq \text{MAX}(CPL, RPL)$  の検査をするので、タスクスイッチの実行を特定の特権レベルに限定することができる。

また、図6・13に示すようにタスクゲートを IDT に定義すれば、割り込みによってタスクスイッチを実行することもできる。割り込みの原因は、INT 命令をはじめとするソフトウェア割り込み、ハードウェア割り込み、内部割り込みの任意の原因が可能である。とにかく、割り込みによって参照された IDT のスロットにタスクゲートが定義されていれば、通常の割り込み処理が実行されるのではなく、タスクスイッチが発生する。したがって、特権レベルの変更によるスタックの切り換え、FLAG、戻りアドレスの CS, IP の PUSH などは、タスクスイッ

使用されない				+0
TSS セクタ				+2
P	DPL	0 0 1 0 1	使用されない	+4
80386 予約領域				+6

図 6・12 タスクゲート

## 6 タスクとタスクスイッチ

チではいっさい実行されない。ただし、第7章で述べる保護例外による割り込みでタスクスイッチが実行されるとき、もしエラーコードがあればタスクスイッチの後、新しいタスクのスタックにエラーコードがPUSHされる。

割り込みによるタスクスイッチのとき、TSS ディスクリプタの B, FLAG の NT, そして、バックリンクセクタの変化は、図6・11に示したCALL命令によるタスクスイッチの場合とまったく同じである。したがって、割り込みによってタスクスイッチした後、NTが1になるから、IRET命令を実行すればバックリンクセクタに記録された以前のタスクに戻ることができる。

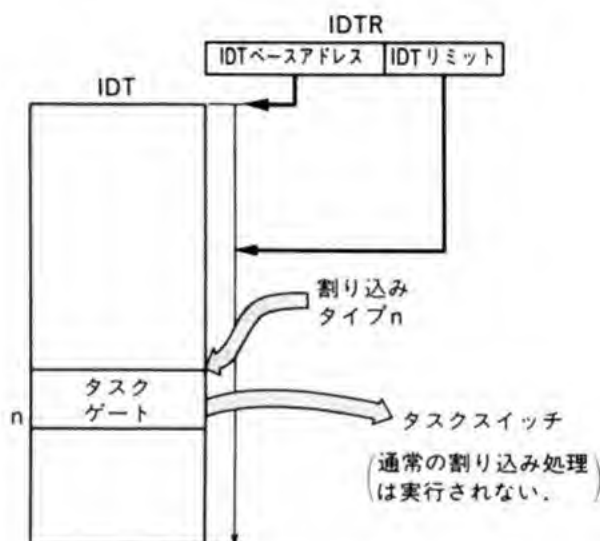


図 6・13 割り込みによるタスクスイッチ

## 6-7 タスクスイッチの例

OS 設計において、タスクスイッチを実行する手続きを一般にディスパッチャと呼ぶ。ディスパッチャは、いくつかのタスクの中から次に実行するタスクを決定し、そのタスクにスイッチする手続きである。図 6・14 に示すように、タスク A とタスク B の間で必要に応じてタスクスイッチをしながら処理を進めるシステムを考える。タスク A のプログラムに属するすべてのセグメントは、LDT\_A という LDT によって管理する。また、タスク B のプログラムで定義されたすべてのセグメントは、LDT\_B という LDT によって管理する。手続きディスパッチャは、タスク A、タスク B の両方から共通に参照される手続きであるから、そのセグメントディスクリプタを GDT に定義すればよい。また、一方のタスクの暴走などによって、ディスパッチャコード、データが書き換えられるようなことがあってはならないから、ディスパッチャのセグメントは特権レベル 0 に定義し、その他のセグメントは特権レベル 3 に定義することになる。

ディスパッチャを引用する方法は、コールゲートを介した CALL 命令を使用することもできるし、また割り込みゲートあるいはトラップゲートを IDT に定義して割り込みによって引用することもできる。外部の周辺装置からデータを入

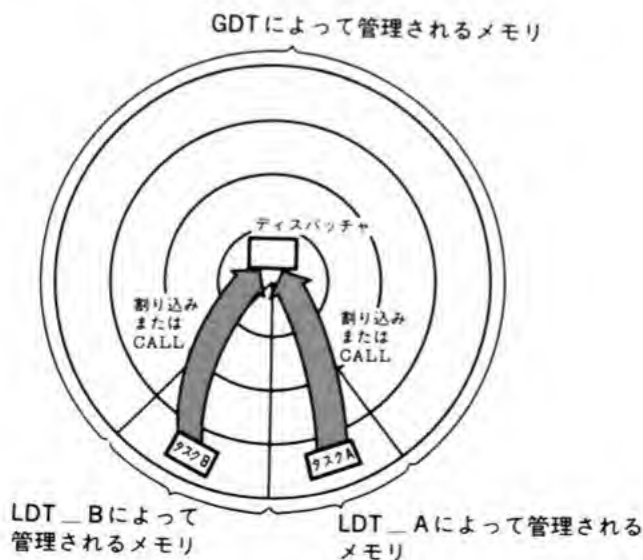


図 6・14 ディスパッチャ

## 6 タスクとタスクスイッチ

```

NAME dispatcher

PUBLIC dispatcher

dispatch_data SEGMENT RW
    task DD 00180000H,00200000H

    task_ptr DW 0
dispatch_data ENDS

stacka STACKSEG 100H
stackb STACKSEG 100H

code SEGMENT EO
    ASSUME DS:dispatch_data

    dispatcher PROC FAR
        CLI
        PUSH DS
        PUSH AX
        PUSH DX
        PUSH BX

        MOV AX,dispatch_data
        MOV DS,AX

        ADD task_ptr,4           ⇒task_ptr に 4 を加算する。
        MOV AX,task_ptr         ⇒task_ptr を 8 で割った余りを求め DX に代入する。
        XOR DX,DX
        MOV BX,8
        DIV BX                  ⇒DX に余りが、そして AX に商が求められる。

        MOV task_ptr,DX
        MOV BX,DX

        JMP task[BX]           ⇒タスクスイッチを実行する。
                                ⇒DX = 0 のとき、タスク tsk4d_tss からタスク
                                exam4d_tss に、また BX = 1 のときタスク
                                exam4d_tss からタスク tsk4d_tss へ
                                スイッチする。

        POP BX
        POP DX
        POP AX
        POP DS
        STI
        IRET

    dispatcher ENDP
code ENDS
NTAINS PRIVILEGED INSTRUCTIONS
END

```

図 6・15 ディスパッチャによるタスクスイッチの例

力しなければならないとき、ディスパッチャを CALL して、他のタスクに制御を渡し、必要なデータが入力されたとき再び制御を取り戻すようなシステムを作ることできるし、また、タイマによって一定時間ごとに割り込みをかけ、ディスパッチャを引用して、タスクを切り換えるようなタイムシェアリングシステムを作ることできる。

図 6・15 に非常に単純化したディスパッチャの例を示す。ハードウェアのタイマによって、一定時間ごとにタイプ 32 の割り込みを発生し、手続き dispatcher に制御を移行するようにする。手続き dispatcher は、配列 task に 2 つのタスクの TSS ディスクリプタのセクタを初期設定し、dispatcher が引用されるたびに、間接 JMP 命令すなわち `JMP task[BX]` が配列 task の要素を交互に参照して、タスクスイッチを実行する。このために、変数 task\_ptr を、dispatcher が引用されるたびに、0 から 4 へ、4 から 0 へというように変化させている。この dispatcher によってスイッチされるタスク A、タスク B の例を、図 6・16、図 6・17 に示す。しかし、いまこの 2 つのプログラムタスク A とタスク B で何をするかということにはあまり興味はない。それらは図 6・16 および図 6・17 に示した以

```

1      NAME task_a
2
3      data SEGMENT RV
4          d1 DW ?
5          d2 DW ?
6          rst DW ?
7      data ends
8
9      stack STACKSEG 100H
10
11     code SEGMENT EO
12         ASSUME DS:data
13         ASSUME SS:stack
14
15     start:
16         MOV d1,2
17         MOV d2,4
18         MOV AX,d1
19         ADD AX,d2
20         MOV rst,AX
21         JMP start
22     code ENDS
23     END start,DS:data,SS:stack

```

図 6・16 タスク A

## 6 タスクとタスクスイッチ

```

1      NAME task_b
2
3      data SEGMENT RW
4          buff DB 100H DUP(?)

5
6      data ENDS
7
8      stack STACKSEG 100H
9
10     code SEGMENT EO
11         ASSUME DS:data
12         ASSUME SS:stack
13
14     start:
15         MOV AX,SEG buff
16         MOV ES,AX
17         MOV DI,OFFSET buff
18         MOV CX,LENGTH buff
19         CLD
20         XOR AL,AL
21     again:
22         STOSB
23         INC AL
24         LOOP again
25
26     JMP start
27 code ENDS
28 END start,DS:data,SS:stack

```

図 6・17 タスク B

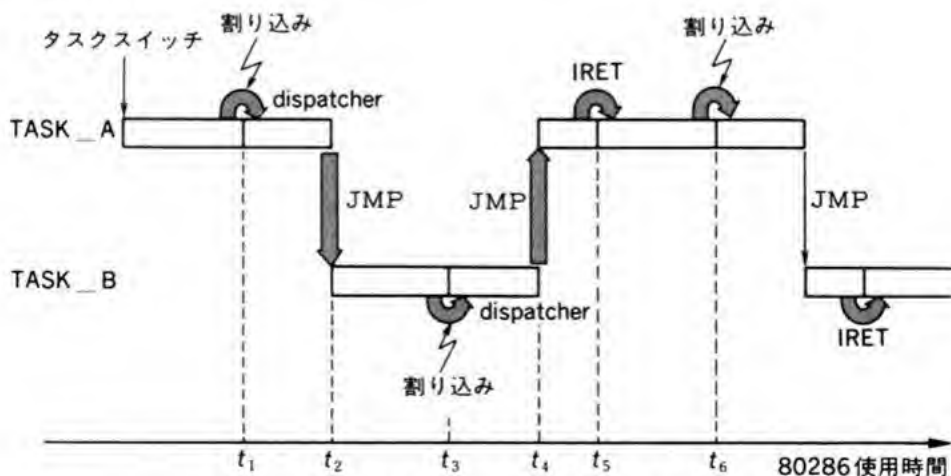


図 6・18 タスクスイッチにおける 80286 処理の流れ

外の任意のプログラムであってもかまわない。

ここでは、このプログラム例で 80286 がどのように使用されるかについて考える。そのために、80286 の実行の時間的な流れを図 6・18 に示す。

タスク A が最初に実行されるものとする。このような初期タスクもタスクスイッチの命令を使用して起動することができる。しばらくして、タイプ 32 の割り込みが発生して、タスク A のメインプログラムから手続き dispatcher に制御が移行する。dispatcher の処理の中で次に実行するタスクを決定し、そのタスクに制御を渡す。この場合は、図 6・15 の `JMP task[BX]` がタスクスイッチの命令になる。ここで、80286 は現在のレジスタの状態をタスク A の TSS に保存し、逆にタスク B の TSS に保存されているレジスタ状態を 80286 のレジスタに展開してから処理を続ける。このとき、タスク B がプログラムのどこから実行するかということは TSS の初期値によって決まる。

タスク B を実行しているとき、再びタイマからタイプ 32 の割り込みが入ると、タスク B のメインプログラムから dispatcher に制御が移行する。ここで、タスク A の場合と同じ手続き dispatcher が引用されているわけであるが、今度の dispatcher はタスク B で動作していることに注意するべきである。すなわち、dispatcher はメモリ上に 1 つしか存在しないにもかかわらず、タスク A で実行される dispatcher とタスク B で実行される dispatcher は、その動作においてまったく無関係である。

このタスク B で実行される dispatcher によって、再びタスク B からタスク A にスイッチされる。このとき、以前、タスク A の TSS に保存された状態が再び 80286 のレジスタに展開されるから、タスク A の処理は、図 6・15 に示す dispatcher のタスクスイッチの命令の次の命令 `POP BX` から再開される。そして、dispatcher の最後の `IRET` 命令によって割り込み処理から、再びメインプログラムに戻る。この後は同様の処理が繰り返し実行される。



## 7. 保護例外

80286 の内部割り込みの中で、保護例外と呼ばれるものがある。保護例外による割り込みでは、スタックに FLAG、戻りアドレスの CS、IP を PUSH し、さらに 1 ワードのエラーコードを PUSH する。保護例外はプロテクトモードにおけるメモリ管理、タスク管理に関する例外割り込みで、基本的にプロテクトモードにおいてのみ作用する。

## 7-1 保 護 例 外

保護例外は不正なディスクリプタ，TSS の参照または特権規則に違反した場合に発生する例外割り込みである．保護例外の種類を表 7・1 に示す．保護の原因によって，割り込みタイプが定義されている．保護例外は，割り込み処理の最初の命令を実行する前に，スタックに図 7・1 に示すような 1 ワードのエラーコードを PUSH すること以外は一般の割り込みと同じである．

エラーコードのビット 0 が 1 のとき，外部からのハードウェア割り込みまたはシングルステップによる割り込みを処理中に，保護例外が発生したことを表す．逆に，ビット 0 が 0 のときは，プログラム中の命令が保護例外の原因になっていることを表す．

エラーコードのビット 1 からビット 15 までは，保護例外の原因となったディスクリプタが存在するテーブルとインデックスを表す．IDT に定義するディスクリプタ（すなわち，ゲート）の最大のインデックスは 255 であるから，エラーコードのビット 1 が 1 のとき，ビット 11 からビット 15 まではすべて 0 である．また，保護例外の原因によっては，エラーコードはすべて 0 の場合もある．

保護例外に対応する IDT のディスクリプタには，割り込みゲート，トラップゲートまたはタスクゲートを定義すればよい．割り込みゲートまたはトラップ

表 7・1 保護例外の種類

割り込み タイプ	保護例外の種類	エラーコード	割り込みゲートまたはトラップ ゲートを使用したときスタック に PUSH される戻りアドレス	リアルモード	プロテ クトモード
8	2重エラー	0	保護例外の原因となった 命令の先頭アドレス	特定条件で作 用する	作用する
10	不正 TSS	原因によって 決まる		作用しない	作用する
11	セグメント不在	原因によって 決まる		作用しない	作用する
12	スタックエラー	原因によって 決まる		作用しない	作用する
13	一般保護エラー	原因によって 決まる		特定条件で作 用する	作用する

ゲートを使用するとき、割り込み手続きによって保護例外を処理することができる。また、タスクゲートを使用するときは保護例外を別のタスクで処理することができる。

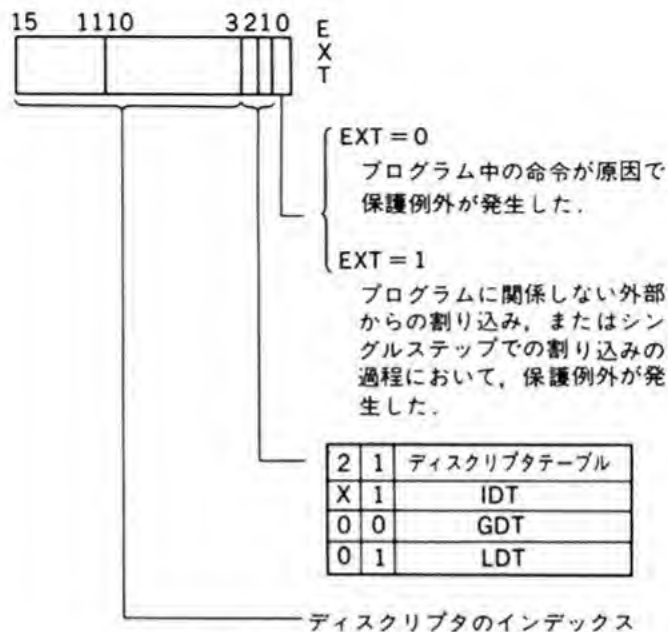


図 7・1 保護例外のエラーコード

## 7-2 スタックエラー

スタックエラーは、SS レジスタを使用したメモリ参照において、セグメントリミットを超えた場合、または SS に代入するセレクトアが P=0 のディスクリプタを指定しているときに発生し、タイプ 12 の割り込み処理を実行する。

PUSH 命令、POP 命令、CALL 命令、割り込みなどの他に、MOV AX, [BP] のような命令によっても、前者の原因でスタックエラーが発生する。このとき、タイプ 12 の割り込み処理を実行する前に、0 のエラーコードが PUSH される。

後者の原因でのスタックエラーは、SS に新しいセレクトアを代入するすべての命令において発生する可能性がある。もちろん、タスクスイッチの命令の中でも、SS に新しいセレクトアを代入する処理が含まれるので、スタックエラーが発生する可能性はある。

タスクスイッチ中にスタックエラーが発生した場合、付録 B のタスクスイッチのシーケンスからわかるように、タイプ 12 の割り込み処理は新しいタスクで実行される。しかし、タスクスイッチの際に発生するスタックエラーは DS キャッシュまたは ES キャッシュにディスクリプタを代入する前に発生し、DS キャッシュ、ES キャッシュは未定義のまま残ってしまう。したがって、割り込み手続きの先頭で

```
MOV AX, DS
MOV DS, AX
MOV AX, ES
MOV ES, AX
```

のような処理を実行して、DS および ES のキャッシュを定義する必要がある。あるいは、タイプ 12 の割り込みにタスクゲートを使用して、スタックエラーを別のタスクで処理してもよい。

特権レベル 0 において発生したスタックエラーは、必ず別のタスクで処理するべきである。そうでなければ、スタックエラーの割り込みにおいて、再び同じスタックセグメントを使用することになるからである。

## 7-3 TSSエラー

TSSエラーは、タスクスイッチ実行中に表7・2に示す状況において発生し、タイプ10の割り込みを実行する。TSSエラーのエラーコードのインデックス部はその原因によって、表に示すように決まる。

IDTのタイプ10のゲートには、必ずタスクゲートを定義し、TSSエラーを別のタスクで処理しなければならない。

表 7・2 TSSエラーを発生する原因

原 因	エラーコード
TSS ディスクリプタのリミット < 43	TSS ディスクリプタ のインデックス
LDT のセクタが不正なディスクリプタを指定する。 または LDT ディスクリプタの P が 0 である。	LDT ディスクリプタ のインデックス
SS のセクタがテーブルリミットを超えている。	SS ディスクリプタの インデックス
SS のディスクリプタにおいて、W = 0	SS ディスクリプタの インデックス
SS のディスクリプタにおいて、DPL ≠ CPL	SS ディスクリプタの インデックス
SS のセクタにおいて、RPL ≠ CPL	SS ディスクリプタの インデックス
CS のセクタがテーブルリミットを超えている。	CS ディスクリプタの インデックス
CS のディスクリプタにおいて、E = 0	CS ディスクリプタの インデックス
CS のディスクリプタにおいて、C = 0 のとき、DPL ≠ CPL	CS ディスクリプタの インデックス
CS のディスクリプタにおいて、C = 1 のとき、DPL > CPL	CS ディスクリプタの インデックス
DS, ES のセクタがテーブルリミットを超えている。	DS または ES ディス クリプタのインデックス
DS, ES のディスクリプタにおいて、E = 1 かつ R = 0	DS または ES ディス クリプタのインデックス

この表において、LDT のセクタ、SS のセクタ、CS のセクタ、DS のセクタ、ES のセクタ、CPL、RPL はすべてタスクスイッチにおいて、新しいタスクの TSS に保存されている値を表す。

## 7-4 Pビットエラー

Pビットエラーは、P=0のディスクリプタを、セグメントキャッシュ、TRキャッシュ、LDTRキャッシュに代入するときに発生し、タイプ11の割り込み処理を実行する。このとき、エラーコードは、原因となったディスクリプタのセクタを表す。

3-5で述べたように、Pビットエラーを利用すれば、1タスク当たり1Gバイトの仮想メモリ空間を実現することができる。しかし、仮想メモリはハードディスクなどの補助記憶装置と実メモリとの間でスワップを実行するため、大きなメモリ空間を必要とするマルチユーザのタイムシェアリングシステムにおいては有効であるが、タイミングの重要な工業用のリアルタイム制御には向かない。この場合、すべてのディスクリプタのPを1にしておけば、Pビットエラーの保護例外は発生しない。

タスクスイッチ中に発生するPビットエラーを、割り込みゲートまたはトラップゲートを使用して処理する場合にも、80286のタスクスイッチ処理の順序から、DS、ESのセクタは定義されているが、DSおよびESのキャッシュにはディスクリプタが定義されていない。したがって、このような状態のPビットエラーをタスクゲートを使用して、別タスクで処理するならば問題はないが、割り込みゲートまたはトラップゲートを使用して、割り込み手続きで処理するときには

```
MOV AX,DS  
MOV DS,AX  
MOV AX,ES  
MOV ES,AX
```

の命令を実行して、DSおよびESのキャッシュを定義しなければならない。

## 7-5 一般保護エラー

一般保護エラーは、スタックエラー、TSS エラー、P ビットエラーそして次に述べる 2 重エラー以外の一般的な保護例外によって発生し、タイプ 13 の割り込み処理を実行する。一般保護エラーの原因としては、たとえばセグメントリミットを超えたメモリ参照、 $E=1$  のセグメントへのデータライト、 $W=0$  のセグメントに対するデータリード、ヌルセクタ (0) をもつ DS、ES を使用したメモリ参照、特権レベルによる保護、不正なディスクリプタの参照などがある。

一般保護エラーのエラーコードは、状況によって 0 またはエラーの原因となったディスクリプタのセクタが、図 7-1 に示した形式でスタックに PUSH される。

### 保護例外の効用

8086 のアセンブリ言語のプログラミングで最初よく犯すうっかりミスは、DS、ES などのセグメントレジスタの初期設定を忘れることである。プログラミングに習熟してくれば、「プログラムの実行開始時には必ずセグメントレジスタの初期設定をする」また「far タイプの手続きの先頭では、セグメントレジスタをスタックに PUSH してから新しい値を代入する」というようにプログラムのスタイルが決まってくるので、上記のようなミスを犯すことはまずない。しかし、何かの都合によってプログラムのスタイルが変わるようなとき、セグメントレジスタの初期設定を忘れることがある。

8086 でやっかいなのは、DS にでたらめな値が入っていてもそれなりに動作してしまうことである。もちろん、まったく別のメモリ空間をデータセグメントとして使用するわけであるから、他のデータを壊すなどして正しい動作はしない。たいていの場合そのプログラムは暴走する。

しかし、プロテクトモードの 80286 では、DS に誤ったデータが入った場合、誤って動作することはまずない。誤った処理を実行する命令は保護によって例外割り込みが発生する。8086 の OS を 80286 のプロテクトモードに移植したとき、それまで発見されなかったバグが発見できた例もある。

## 7-6 2重エラー

**2重エラー**は、1つの命令において2重の保護が働いたときに発生し、タイプ8の割り込みを実行する。たとえば、一般保護エラーが発生して、その割り込み処理においてスタックにFLAG、戻りアドレスのCS、IPそしてエラーコードをPUSHしたところが、スタックのリミットを超えたような場合に、2重エラーが発生する。

2重エラーに対する処理は、必ず別タスクで実行するべきである。さもないと3重のエラーが発生する可能性がある。しかし、80286には、3重エラーに対する保護例外はもっていない。もし、2重エラーによるタイプ8の割り込みの過程で再び保護例外が発生すれば、80286は**シャットダウン状態**になり、すべての機能を停止する。シャットダウン状態は、80286がHLT命令を実行した後の**ホルト状態**と同じである。ただし、シャットダウン状態とホルト状態は、アドレスバスのA<sub>1</sub>端子のレベルによって、ハードウェア的に見分けることができる。シャットダウン状態のときはA<sub>1</sub>=Lowであり、ホルト状態のときはA<sub>1</sub>=Highである。

また、シャットダウン状態から抜け出すためには、NMI端子に割り込み信号を入れる方法と、80286をリセットする方法がある。

## 7-7 例外処理と再実行

保護例外は、エラーコードを PUSH する以外は、通常の割り込みと同様である。保護例外に使用するゲートは、例外割り込みが発生する状況によって割り込みゲート、トラップゲートそしてタスクゲートのどれかを使用することができる。図7-2(a), (b), (c)に割り込みゲートまたはトラップゲートを使用して、同じ特権レベルの割り込み処理に移行する場合、より高い特権レベルの割り込み処理に移行する場合、タスクゲートを使用して、別のタスクにタスクスイッチした場合のそれぞれのスタックの様子を示す。

割り込みゲートまたはトラップゲートを使用したとき、スタックに PUSH される戻りアドレスは、保護例外の原因となった命令の先頭アドレスである。また、

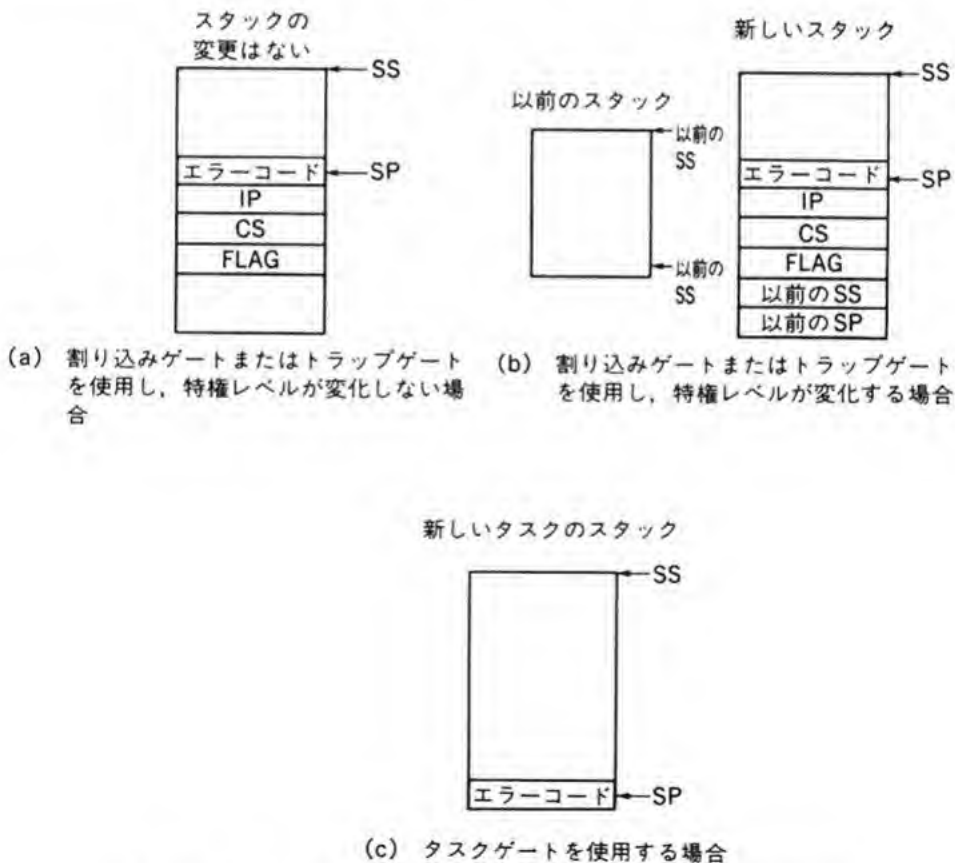


図 7・2 保護例外によって、スタックに PUSH されるエラーコード

## 7 保 護 例 外

タスクゲートを使用した場合、TSS のバックリンクセクタで指定される以前のタスクの TSS には、保護例外の原因となった命令の先頭アドレスを表す CS および IP が保存される。したがって、保護例外の原因を取り除いてから

```
ADD SP, 2
```

```
IRET
```

を実行することによって、保護例外が発生した命令を再実行することができる。

しかし、次の命令は例外的に再実行できない。

(1) セグメントリミットを超えることによる一般保護エラーが発生した REP プリフィックス付きのストリング命令。

(2) ディスクリプタの W が 0 (リードのみ) のセグメントにライトした、ADC 命令、SBB 命令、RCL 命令、RCR 命令。

(3) ディスクリプタの W が 0 (リードのみ) のセグメントにライトした XCHG 命令。

(1) の場合、ストリング命令でカウンタとして使用する CX の更新をしないまま例外割り込みに移行するため、IRET 命令を実行しても繰り返し処理の途中から実行を再開することにならない。(2) の場合は、保護例外が発生するのは演算が終了してからであるから、CF の値は変化してしまう。したがって、IRET 命令によって再実行しても、最初の実行と同じ環境では実行できないからである。(3) の XCHG 命令が再実行できない理由も、1 度目の実行によって他方のオペランドに指定したレジスタの内容が変化してしまうからである。

## 8. 80286のハードウェア

80286 のハードウェア構成は、8086 に比較して決して複雑ではない。しかし、80286 のバスは、8086 のようにアドレスバスとデータバスが共通の端子を使用することではなく、より効率のよいバスサイクルを実現している。また、バスサイクルは2プロセッサクロックで実行され、8086 のバスサイクル速度と比較して2倍になっている。CPU のクロックが高速になるとき問題になるのがメモリ、I/O の応答速度である。CPU のクロックを速くしても、参照するメモリの応答速度が遅いためには意味がない。では、応答速度の速いメモリを使用すればよいが、高価である。また、80286 をプロテクトモードで使用するようなシステムでは、1~2M バイト以上の大容量のメモリを使用することが多い。高速で80286 を効率よく動作させるシステムを作るために高価なメモリを大量に使用することは経済的に問題がある。そこで、80286 では、比較的安価なメモリを使用しても十分なスピードのシステムを作れるようになっている。

## 8-1 CPU モジュールの構成

〔1〕 電力供給とシステムクロック 80286 の端子配置図は図 1・5 に示した。表 8・1、表 8・2 に各端子の信号と働きを示す。80286 への電力供給は、図 8・1 に示すように、 $V_{cc}$  に +5V の電圧を与え、 $V_{ss}$  を信号グラウンドに接続する。また、CAP 端子と信号グラウンド間には、 $0.047\mu F \pm 20\%$  12V のコンデンサを接続する。これは 80286 の内部サブストレートバイアスのフィルタとなる。 $V_{cc}$ 、 $V_{ss}$  端子はそれぞれ複数あり、内部では共通であるが、これらはすべて接続するようにする。

CLK 端子にはシステムクロックを供給する。80286 の内部回路は、このシステムクロックを 2 分周したプロセッサクロックに同期して動作する。すなわち、8MHz で動作する 80286 には、16MHz のシステムクロックを供給する。このシステムクロックは、図 8・1 に示すように、クロックジェネレータ 82284 によって作られる。

表 8・1 80286 端子一覧表

信号と記号	信号の方向	説 明
CLK (システムクロック)	入力	システムクロックを入力する。80286 内部の分周回路によって CLK を 2 分周して、プロセッサクロック (PCLK) が作られる。80286 内部の各回路は PCLK に同期して動作する。
$D_0-D_{15}$ (データバス)	入出力	メモリまたは I/O インタフェースとの間でデータを入出力するために使用する。
$A_0-A_{23}$ (アドレスバス)	出力	メモリまたは I/O インタフェースにアドレスを出力するために使用する。
$\overline{BHE}$ (バスハイイネーブル)	出力	データの入出力において、データバスの上位バイト ( $D_8-D_{15}$ ) を使用するとき Low が出力され、奇数アドレスのメモリ、I/O に対してデータの入出力が行われることを表す。
$\overline{S_0}, \overline{S_1}$ (バスサイクルステータス)	出力	$COD/\overline{INTA}$ 、 $M/\overline{IO}$ と組み合わせ、80286 のバスサイクルの状態が表 8・2 のように表される。
$M/\overline{IO}$ (メモリ、I/O)	出力	メモリに対するバスサイクルのとき High を出力し、I/O に対するバスサイクルのとき Low を出力する。
$COD/\overline{INTA}$ (コード、割り込みア クノリッジ)	出力	$M/\overline{IO} = \text{High}$ の状態において $COD/\overline{INTA} = \text{High}$ はコードフェッチを表す。また $M/\overline{IO} = \text{Low}$ の状態において $COD/\overline{INTA} = \text{Low}$ は割り込みアクノリッジを表す。

## 8-1 CPU モジュールの構成

信号と記号	信号の方向	説 明
$\overline{\text{LOCK}}$ (バスロック)	出力	LOCK 付き命令、XCHG 命令、割り込みアクノリッジ、ディスクリプタテーブル参照によって実行されるメモリ参照のバスサイクルにおいて Low を出力する。システムバス制御に利用する。
$\overline{\text{READY}}$ (バスレディ)	入力	バスサイクル終了のタイミングを制御する。 $T_c$ の最後で $\overline{\text{READY}}$ が Low のときバスサイクルは終了し、 $\overline{\text{READY}}$ が High のとき再び $T_c$ を繰り返す。
HOLD (バスホールドリクエスト)	入力	ローカルバスの所有権を制御する。他のハードウェアが 80286 の HOLD を High にすることによって 80286 はバスの各端子を OFF 状態にし、その確認信号として HLDA を High にする。HLDA が High になっている間、80286 以外のハードウェアがローカルバスを制御することができる。
HLDA (ホールドアクノリッジ)	出力	
INTR (割り込みリクエスト)	入力	割り込みコントローラ 8259A からの割り込み要求信号を入力する。割り込み要求信号が 80286 内部でマスクされていないければ割り込みアクノリッジサイクルを実行し、8259A から割り込みタイプをリードし、割り込みを発生する。
NMI (ノンマスクابل割り込みリクエスト)	入力	エッジトリガの信号を入力することによって、80286 は無条件にタイプ 2 の割り込みを発生する。
PEREQ (80287 リクエスト)	入力	80286 はメモリ参照を伴う ESC 命令を実行したとき、数値演算プロセッサ 80287 が出力する PEREQ 信号を入力可能にする。実際に 80287 から PEREQ 端子に High の信号が送られたとき、80286 は 80287 が要求するバスサイクルを実行する。また、その最初のバスサイクルにおいて、80287 へのアクノリッジ信号として PEACK に Low のパルスを出力する。
PEACK (80287 アクノリッジ)	出力	
$\overline{\text{BUSY}}$ (80287 ビジー)	入力	数値演算コプロセッサ 80287 の動作状態を 80286 に知らせる。 $\overline{\text{BUSY}}$ 端子が Low になっている状態では再び $\overline{\text{BUSY}}$ 端子が High になるまで 80286 は ESC 命令、WAIT 命令を実行しない。
$\overline{\text{ERROR}}$ (80287 エラー)	入力	数値演算コプロセッサ 80287 から 80286 に例外処理を要求するために使用する。80286 においてタイプ 16 の割り込みを発生する。
RESET (システムリセット)	入力	80286 の内部状態を初期化する。
$V_{ss}$ (信号グラウンド)	入力	電圧 0 [V] に接続する。
$V_{cc}$ (電源)	入力	+5 [V] の電源を供給する。
CAP (サブストレートフィルタキャパシタ)	入力	CAP には $V_{ss}$ との間に $0.047 \mu\text{F} \pm 20\%$ 12 V のコンデンサを接続する。これは 80286 内部のサブストレートバイアスジェネレータ出力のフィルタとなる。

## 8 80286 のハードウェア

82284 の端子配置図とブロック図を図 8・2 に示す。X<sub>1</sub> と X<sub>2</sub> の間に水晶振動子を接続することによって、内部のクリスタルオシレータが発振し、CLK 端子にクロックを出力する。また、内部のクリスタルオシレータを使用せず、外部のオシレータの出力を EFI 端子に接続し、クロックを得ることができる。内部のオシレータを使用するか、EFI 端子からの入力信号を使用するかは、F/ $\overline{C}$  によって決める。F/ $\overline{C}$  を Low にしたとき、内部のオシレータの出力がクロックとして CLK 端子に出力され、F/ $\overline{C}$  を High にしたとき、EFI 端子の入力信号がクロックとして CLK 端子に出力される。また、82284 の PCLK 端子には CLK 端子に出力されるシステムクロックを 2 分周したクロックが出力されている。この PCLK は、80286 のプロセッサクロックに同期し、82284 の PCLK が High のとき CLK の立ち下がりがプロセッサクロックの  $\phi 1$  の終了のタイミングになる。

〔2〕 **RESET と  $\overline{READY}$**  82284 はシステムクロックを 80286 に供給する

表 8・2 ステータス信号とバスサイクル

COD/ $\overline{INTA}$	M/ $\overline{IO}$	$\overline{S_1}$	$\overline{S_0}$	バスサイクル
0	0	0	0	割り込みアクノリッジ
0	0	0	1	この組み合わせの信号は発生しない。
0	0	1	0	この組み合わせの信号は発生しない。
0	0	1	1	アイドル状態
0	1	0	0	A1=1 の場合 ホルト状態 A1=0 の場合 シャットダウン状態
0	1	0	1	メモリデータリード
0	1	1	0	メモリデータライト
0	1	1	1	アイドル状態
1	0	0	0	この組み合わせの信号は発生しない。
1	0	0	1	I/O リード
1	0	1	0	I/O ライト
1	0	1	1	アイドル状態
1	1	0	0	この組み合わせの信号は発生しない。
1	1	0	1	コードフェッチ
1	1	1	0	この組み合わせの信号は発生しない。
1	1	1	1	アイドル状態

だけではなく、システムクロックに同期した **READY** 信号と **RESET** 信号を供給する。RESET 信号は図 8・3 に示すように、16 CLK 以上の長さをもつパルスをもつ 80286 に入力して、80286 の内部状態を初期化する。80286 に電源を投入するときには、必ず RESET 信号をも供給しなければならない。ただし、電源投入時の RESET 信号は 50 ms 以上の間 High にしてから、Low に下がるようなパルスにする。これは、CAP 端子に接続するコンデンサの充電時間である。

RESET 信号が Low になるとき、80286 の信号出力端子は図 8・3 に示すように High または Low に決まり、RESET 信号が Low になってから、約 38 CLK 後に最初のバスサイクルが実行される。このとき 80286 の各レジスタの値は表 8・3 に示すように決まる。したがって、リセット後の 80286 はリアルモードで始まり、物理アドレスが **0FFFFFF0H** 番地のメモリから最初の命令をフェッチして、実行する。このアドレスには、一般にシステムの初期設定を実行するプログラムに制御を移行する JMP 命令を定義する。

80286 に電源を供給したとき確実にリセットするために、82284 の **RES** 端子に図 8・1 に示したような RC 回路を接続する。RES 端子の入力電圧は、82284 内部のシュミットトリガ回路によって波形整形された後、CLK の立ち下がり同期して RESET 端子に出力される。

82284 から 80286 に供給される **READY** 信号によって、80286 のバスサイクル

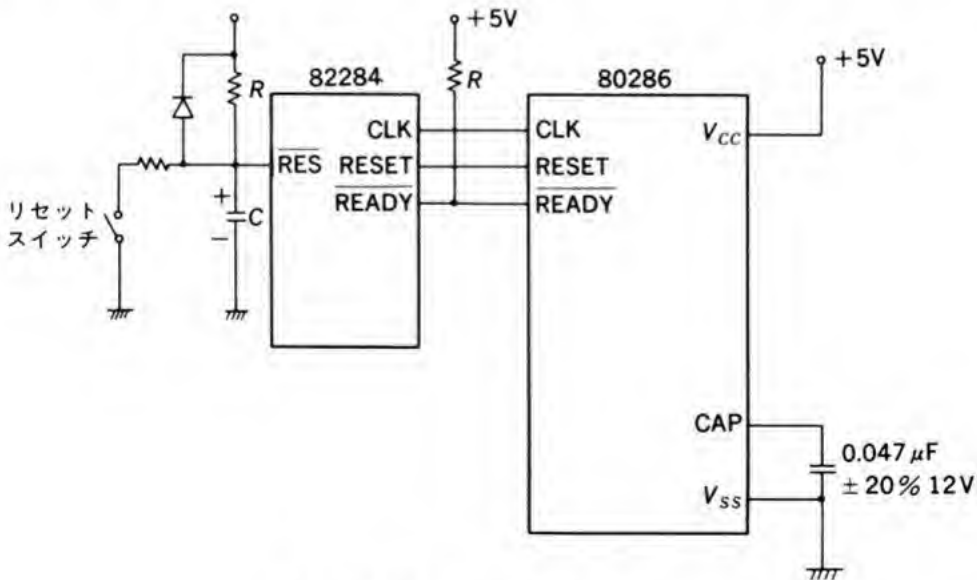
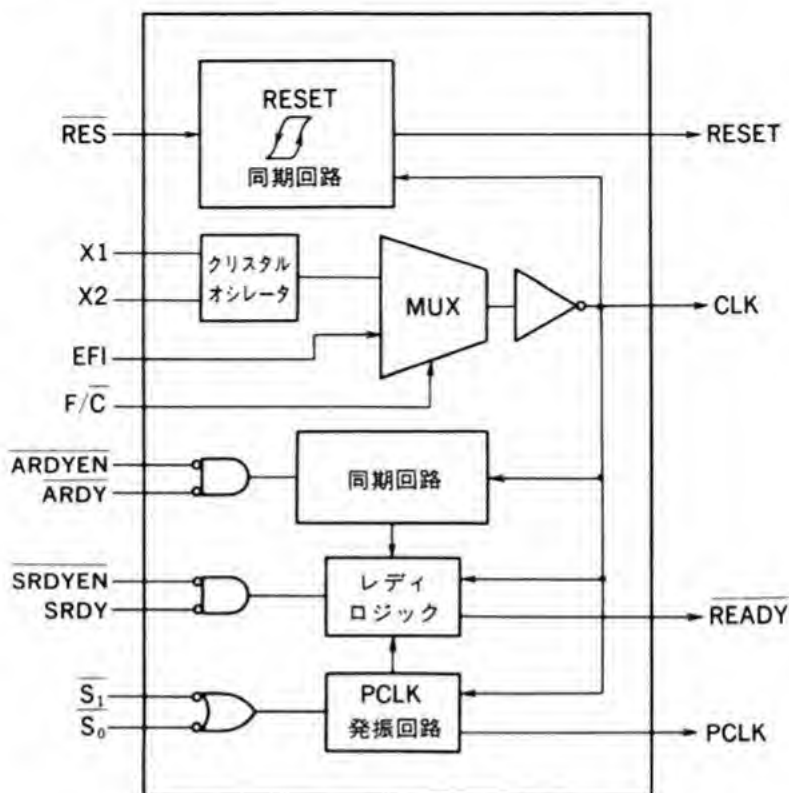
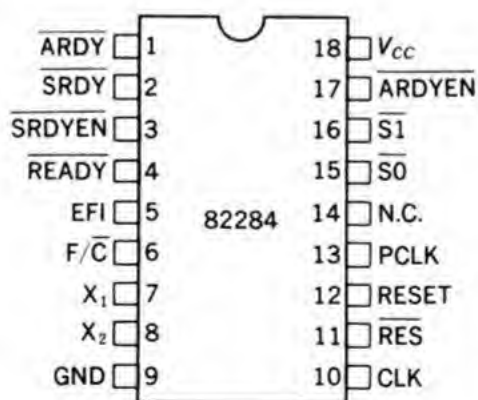


図 8・1 電力供給と 82284 の接続



(a) 82284 ブロック図



(b) 82284 ピン配置図

図 8・2 82284 クロックジェネレータ

(iAPX 286 Hardware Reference Manual  
(Order No.210760-001), A-75, Fig1, Fig2 より引用)

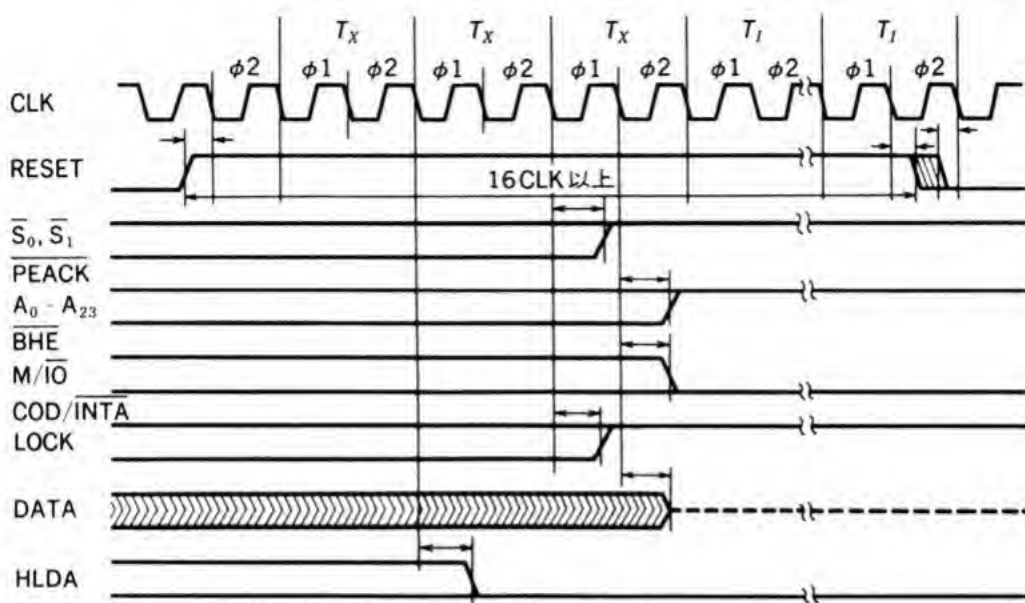


図 8・3 リセット信号

(iAPX 286 Hardware Reference Manual  
(Order No.210760-001), 3-68, Fig 3-69 より引用)

表 8・3 80286 リセット後のレジスタの状態

レジスタ名	初期値
FLAG	0002H
MSW	0FFF0H
IP	0FFF0H
CS	0F000H
DS	0000H
SS	0000H
ES	0000H
CS キャッシュ	ベースアドレス = 0FF0000H, リミット = 0FFFFH
DS キャッシュ	ベースアドレス = 000000H, リミット = 0FFFFH
SS キャッシュ	ベースアドレス = 000000H, リミット = 0FFFFH
ES キャッシュ	ベースアドレス = 000000H, リミット = 0FFFFH
IDTR	ベースアドレス = 000000H, リミット = 03FFH

の長さが決まる。図 8・4 に示すように、80286 のバスサイクルは、 $T_s$ 、 $T_c$  の 2 つのプロセッサクロックによって構成されるが、 $T_c$  の最後に  $\overline{\text{READY}}$  信号を High にすることによって、 $T_c$  の後にもう 1 つの  $T_c$  を挿入し、バスサイクルを

延長することができる。T<sub>c</sub>の最後に  $\overline{\text{READY}}$  信号を Low にしたとき、バスサイクルは終了する。

82284 の  $\overline{\text{READY}}$  端子はオープンコレクタであるから、図 8・1 に示したようにプルアップ抵抗を接続しなければならない。また、このため 80286 の  $\overline{\text{READY}}$  端子には、82284 の  $\overline{\text{READY}}$  以外の他のレディ信号をワイヤードオアで接続することが可能である。82284 は  $\overline{\text{SRDY}}$ 、 $\overline{\text{ARDY}}$  の 2 つの入力端子の信号から、CLK に同期した  $\overline{\text{READY}}$  を作る。 $\overline{\text{SRDY}}$ 、 $\overline{\text{ARDY}}$  はそれぞれ、イネーブル端子として  $\overline{\text{SRDYEN}}$ 、 $\overline{\text{ARDYEN}}$  の 2 つの端子をもつ。これらのイネーブル端子を Low にしたときにのみ、図 8・2(a) のブロック図からわかるように、 $\overline{\text{SRDY}}$ 、 $\overline{\text{ARDY}}$  端子からの入力信号がそれぞれ有効に働く。

図 8・5 に  $\overline{\text{SRDY}}$  端子からの入力信号と  $\overline{\text{READY}}$  端子の出力信号の関係を示す。CLK の立ち下がりにおいて、82284 に入力される  $\overline{\text{S}}_0$ 、 $\overline{\text{S}}_1$  の少なくとも一方が Low であれば、82284 は  $\overline{\text{READY}}$  端子をハイインピーダンスの状態にする。82284 の  $\overline{\text{READY}}$  端子はプルアップされているから、バスサイクルの最初に  $\overline{\text{READY}}$  を一度 High にすることができる。この後、 $\overline{\text{S}}_0$ 、 $\overline{\text{S}}_1$  の両方が High になっている状態で、PCLK が High の状態における CLK の立ち下がりにおいて、82284 は  $\overline{\text{SRDYEN}} + \overline{\text{SRDY}}$  の入力信号を調べ、 $\overline{\text{SRDYEN}} + \overline{\text{SRDY}}$  が Low であれば、 $\overline{\text{READY}}$  端子に Low を出力する。

また図 8・6 に  $\overline{\text{ARDY}}$  の入力信号と  $\overline{\text{READY}}$  の出力の関係を示す。82284 は CLK の立ち下がりにおいて、 $\overline{\text{ARDYEN}} + \overline{\text{ARDY}}$  の状態を調べ、それが Low であれば、次の PCLK が High の状態における CLK の立ち下がりにおいて  $\overline{\text{READY}}$  を Low にする。したがって、 $\overline{\text{ARDY}}$  を使用するとき、 $\overline{\text{READY}}$  は最少でも 1 CLK だけ Low になるタイミングが遅れるが、プロセッサクロックの  $\phi 1$  の立ち下がりに内部で同期して出力されるため、 $\overline{\text{ARDY}}$  に入力するレディ信号は非同期でもかまわない。 $\overline{\text{SRDY}}$ 、 $\overline{\text{ARDY}}$  のどちらかにレディ信号が供給されることによって、 $\overline{\text{READY}}$  に Low の信号を出力する。

〔3〕 **コマンド信号の生成** 80286 がバスサイクルを実行するとき、メモリ、I/O インタフェースは対応する動作を実行する。このために、メモリまたは I/O インタフェースにコマンド信号と呼ぶ信号を供給する。しかし、80286 はコマンド信号を直接に出力せず、代わりにステータス信号  $\overline{\text{S}}_0$ 、 $\overline{\text{S}}_1$  と M/ $\overline{\text{IO}}$  信号を出力する。80286 は  $\overline{\text{S}}_0$ 、 $\overline{\text{S}}_1$ 、M/ $\overline{\text{IO}}$  の 3 つの信号の組み合わせによって、表 8・4 に示

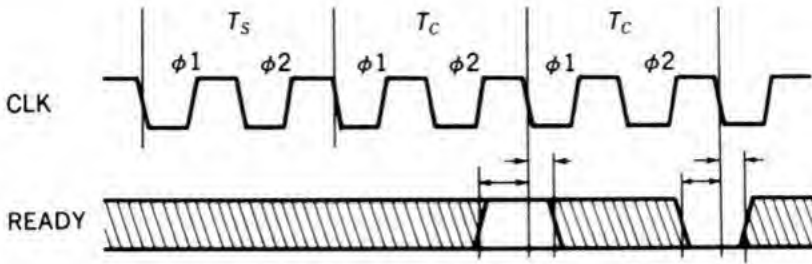


図 8・4 バスサイクルの終了

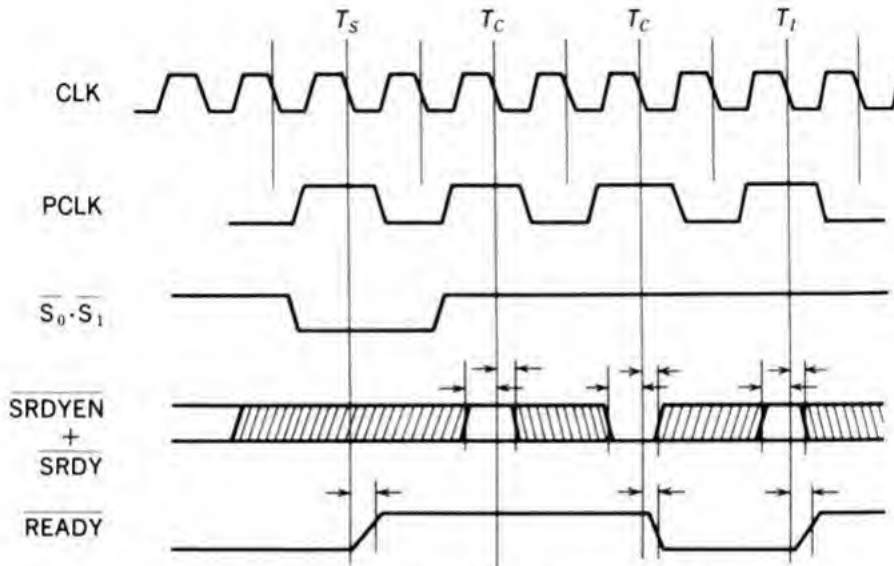


図 8・5 SRDY 制御による 1 ウェイトステートの挿入

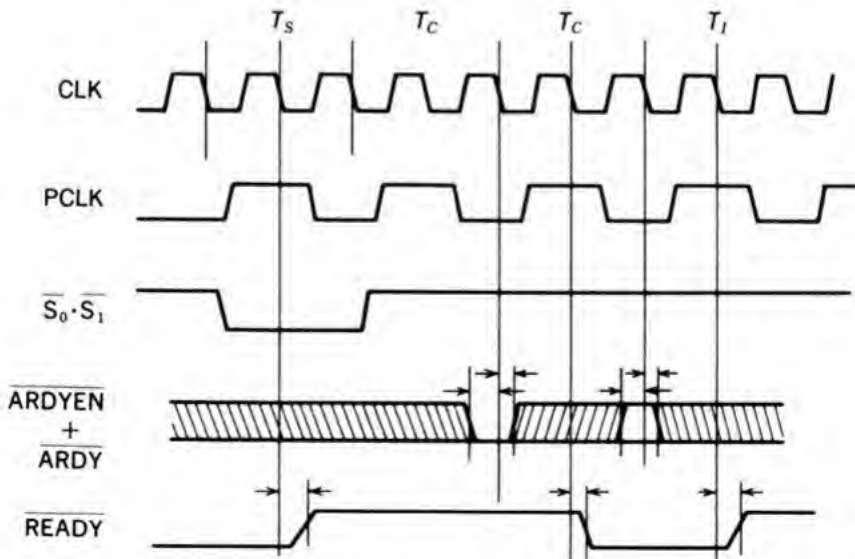


図 8・6 ARDY 制御による 1 ウェイトステート

## 8 80286 のハードウェア

表 8・4 バスサイクルの種類

M/I $\bar{O}$	$\bar{S}_1$	$\bar{S}_0$	バス動作
0	0	0	割り込みアクノリッジ
0	0	1	I/O リード
0	1	0	I/O ライト
0	1	1	アイドル状態
1	0	0	ホルト ( $A_1 = 1$ の場合) またはシャットダウン ( $A_1 = 0$ の場合)
1	0	1	メモリリード
1	1	0	メモリライト
1	1	1	アイドル状態

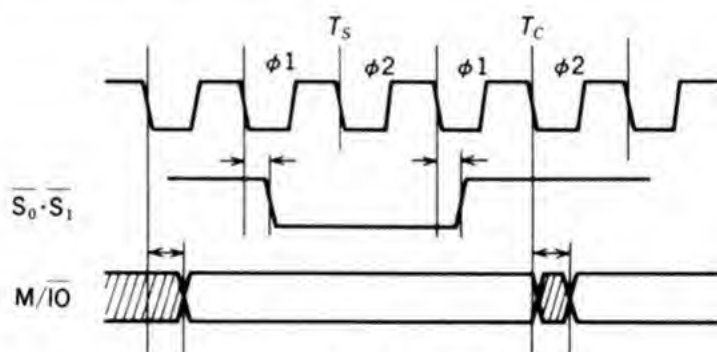


図 8・7 ステータス信号の出力タイミング

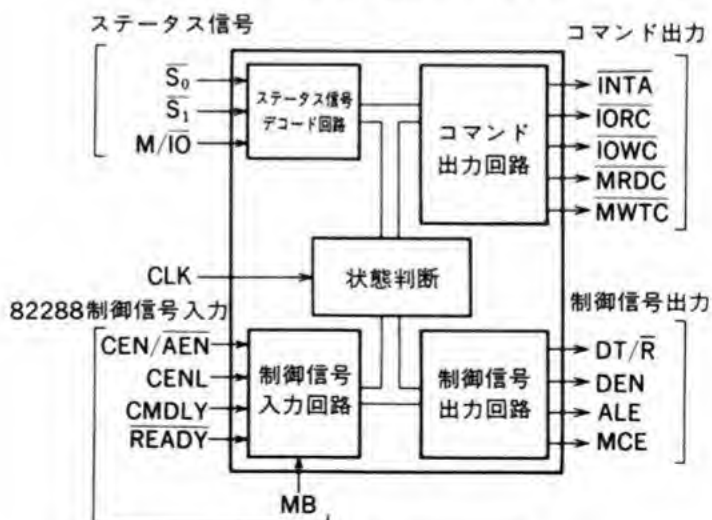


図 8・8 82288 バスコントローラ

(iAPX 286 Hardware Reference Manual  
(Order No. 210760-001), A-83, Fig. 1 より引用)

すように実行するバスサイクルの種類を表現する。また、 $\overline{S_0}$ ,  $\overline{S_1}$ ,  $\overline{M/\overline{IO}}$  が出力されるタイミングを図 8・7 に示す。

しかし、このステータス信号、 $\overline{M/\overline{IO}}$  信号を直接にメモリ、I/O インタフェースに供給することはできない。ステータス信号と  $\overline{M/\overline{IO}}$  信号からコマンド信号を作るために、図 8・8 に示す **バスコントローラ 82288** を使用する。82288 は  $\overline{S_0}$ ,  $\overline{S_1}$ ,  $\overline{M/\overline{IO}}$  の信号をデコードして、割り込みアクノリッジ信号  $\overline{INTA}$ , I/O リードコマンド  $\overline{IORC}$ , I/O ライトコマンド  $\overline{IOWC}$ , メモリリードコマンド  $\overline{MRDC}$ , メモリライトコマンド  $\overline{MWTC}$  を出力する。これらのコマンド信号の出力タイミングは  $\overline{CMDLY}$  信号によって、またコマンド信号の終了タイミングは  $\overline{READY}$

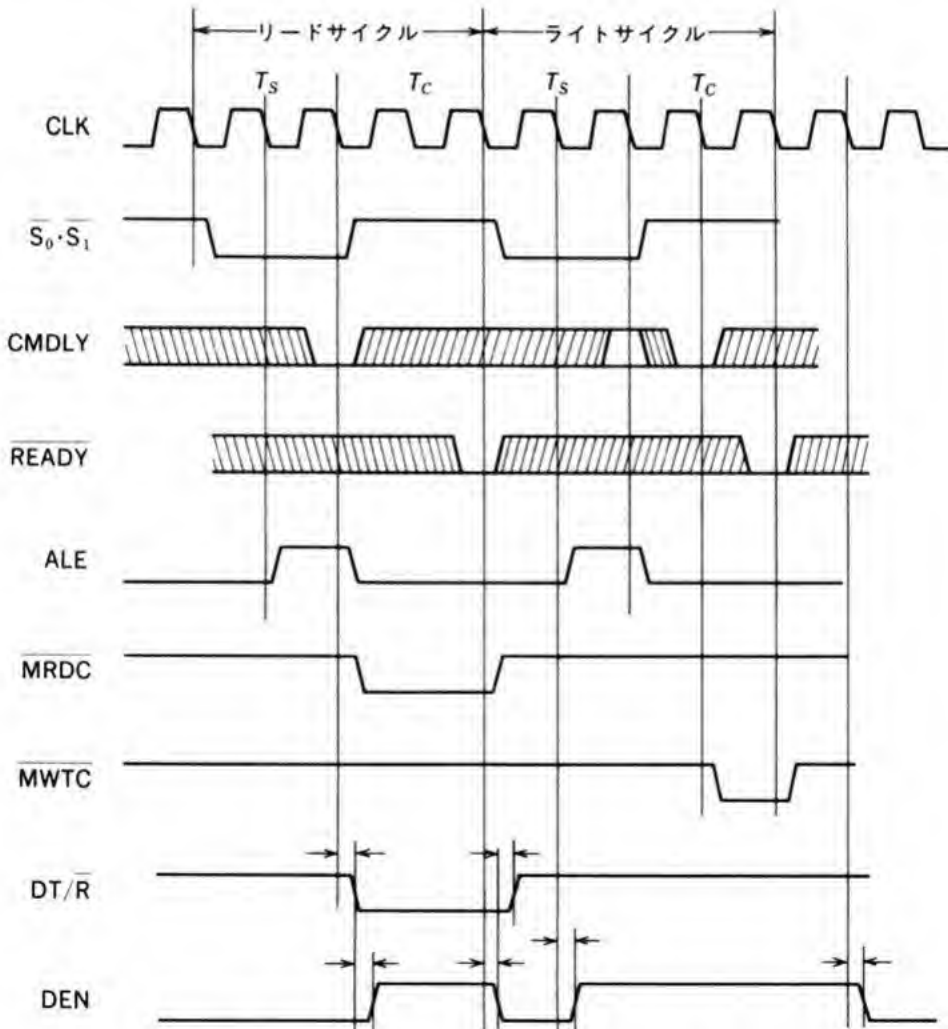


図 8・9 82288 コマンド信号出力タイミング

## 8 80286 のハードウェア

信号によって制御できる。

メモリリードサイクルとメモリライトサイクルが連続して実行されるときタイミグチャートを図 8・9 に示す。最初のリードサイクルにおける  $\overline{\text{MRDC}}$  は、 $T_s$  の立ち下がりにおいて  $\text{CMDLY}$  が Low であるため  $T_c$  において出力されている。次のライトサイクルにおける  $\overline{\text{MWTC}}$  は、 $\text{CMDLY}$  が  $T_s$  の立ち下がりにおいて High で、次の  $T_c$  の  $\phi 1$  の立ち下がりにおいて Low となっているため、1 CLK だけ遅れて出力される。このような  $\text{CMDLY}$  の制御は  $\overline{\text{MRDC}}$ 、 $\overline{\text{MWTC}}$  と同様に  $\overline{\text{IORC}}$ 、 $\overline{\text{IOWC}}$ 、 $\overline{\text{INTA}}$  に対しても有効に働く。CMDLY の制御の必要がなければ、CMDLY 端子を GND (接地) に接続すればよい。

82288 は、コマンド信号の他にアドレスラッチ回路へのストローブ信号 ALE、データバstransシーバへの制御信号  $\text{DT}/\overline{\text{R}}$ 、DEN を出力する。

〔4〕 80286 CPU 構成 80286 を使用した CPU 構成は、80286 を中心として、クロックジェネレータ 82284、バスコントローラ 82288、アドレスラッチ、データバstransシーバによって、図 8・10 のように構成される。

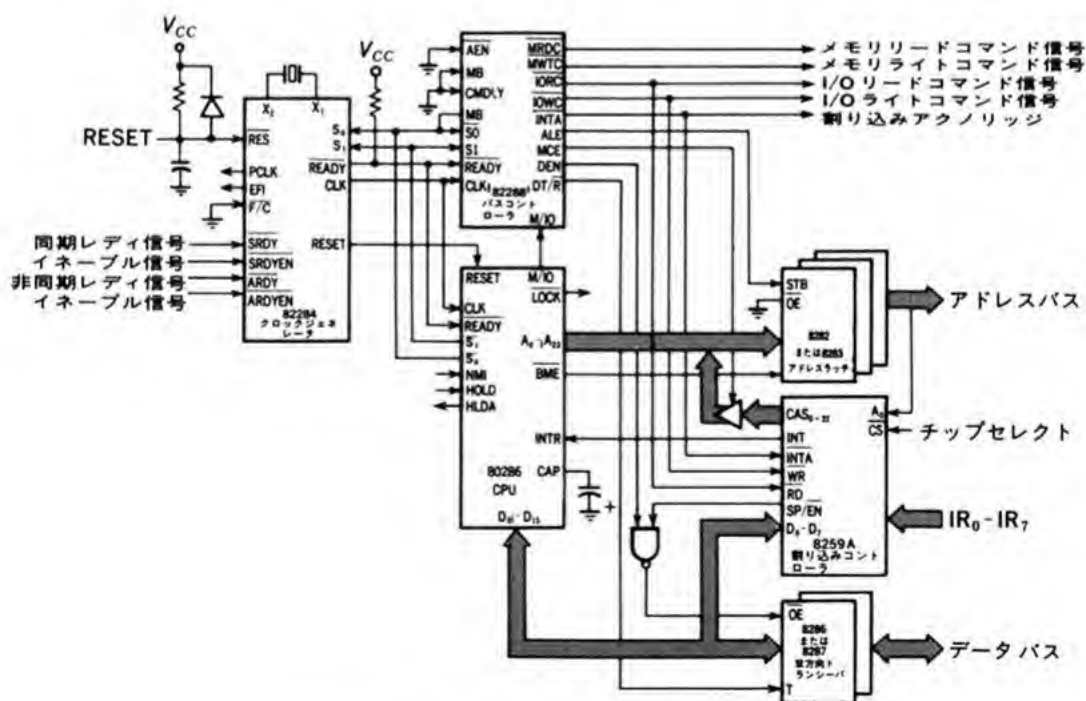


図 8・10 80286 基本構成  
(iAPX286 Hardware Reference Manual  
(Order No.210760-001), A-34, Fig 31 より引用)

## 8-2 メモリインタフェース

〔1〕 基本メモリ構成 80286 のバス信号とメモリとの接続は、基本的には図 8・11 に示すように 8086 の場合と同じである。すなわち、メモリは偶数アドレスのバンクと奇数アドレスのバンクに分割し、偶数バンクのメモリはデータバス  $D_0-D_7$  に接続する。また、奇数バンクのメモリはデータバス  $D_8-D_{15}$  に接続する。アドレスバスには  $A_1$  から  $A_{23}$  の信号を使用し、 $A_0$  は偶数バンクのチップイネーブル信号として使用する。また、奇数バンクのチップイネーブル信号には  $\overline{BHE}$  を使用する。このように接続することによって、 $A_0=0$ 、 $\overline{BHE}=1$  のとき偶数バンクのメモリだけが有効に動作する。また、 $A_0=1$ 、 $\overline{BHE}=0$  のときは奇数バンクのメモリだけが有効に動作する。さらに  $A_0=0$ 、 $\overline{BHE}=0$  のとき、偶数バンク、奇数バンク両方のメモリが有効に動作する。

図 8・11 をコマンド信号も含めて書き換えると、図 8・12 のようになる。図において、アドレスラッチの  $STB$  (ストローブ) 信号にバスコントローラ 82288 の  $ALE$  を供給しているが、図 8・13 に示すように、 $ALE$  信号は  $T_s$  の  $\phi 2$  に出力されるので、アドレスバスにアドレス信号が供給されるのは、 $T_s$  の  $\phi 2$  から、次のバスサイクルの  $T_s$  の  $\phi 1$  の終わりまでである。メモリからデータをリードする場合について考えると、メモリは  $T_c$  の終わりから図 8・13 に示すセットアップ

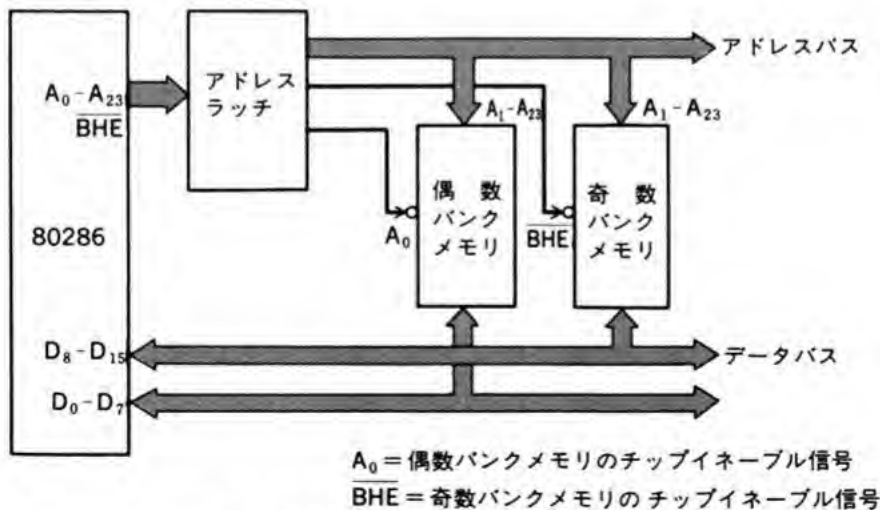


図 8・11 メモリとアドレスバス、データバスの接続

## 8 80286 のハードウェア

タイム以前に、データをデータバスに出力すればよい。したがって、図に示す  $T_{ad}$  はメモリ応答の最大余裕時間であり、 $t_{ad}$  より速い応答速度をもつメモリを使用すれば、問題なくデータをリードすることができる。

〔2〕 特別なストロープ信号を使用したメモリ構成      しかし、80286 自体はアドレス信号を  $T_s$  より以前に出力しているのであるから、図 8・12 に示した回路では、 $T_s$  の  $\phi 2$  以前に出力されているアドレス信号を無駄にしている。アドレスラッチの STB 信号をもう少し早いタイミングで供給できれば、図 8・13 に示した

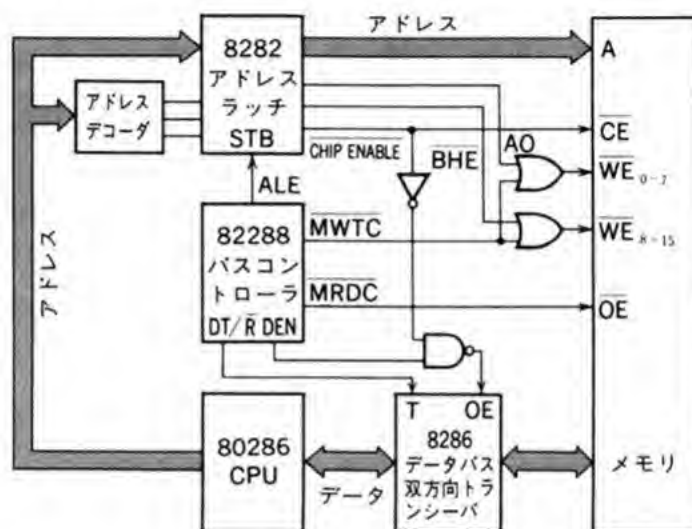


図 8・12 基本的メモリ構成  
(iAPX 286 Hardware Reference Manual  
(Order No.210760-001), 4-7, Fig 4-1 より引用)

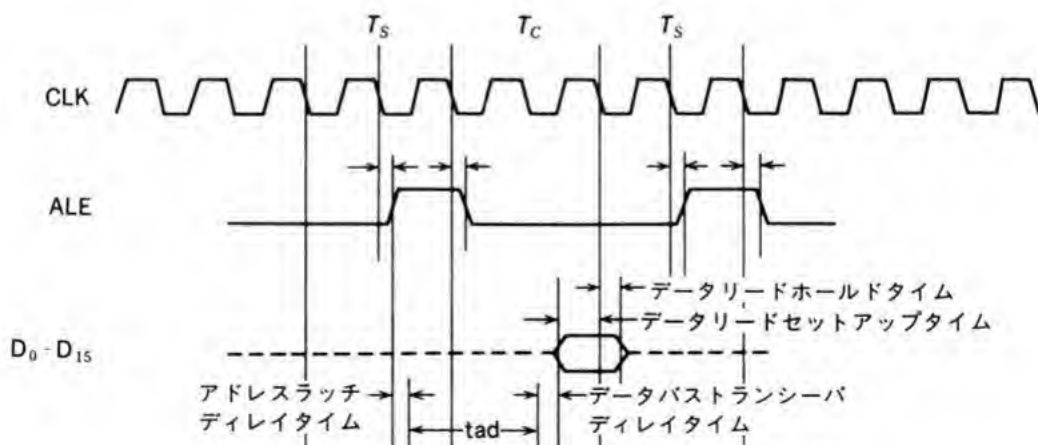


図 8・13 ALE をアドレスラッチのストロープ信号として  
使用した場合のメモリ応答最大余裕時間

## 8-2 メモリインタフェース

$t_{ad}$  より遅い応答スピードをもつメモリを使用することができる。図 8・14 に示すメモリ構成では、82288 が出力する ALE 信号を使用せずに、 $\overline{MRDC} \cdot \overline{MWTC} = 0$  の条件をアドレスラッチのストロブ信号として使用する。

メモリに対するバスサイクルを実行するとき、82288 は遅くとも  $T_s$  の最初に  $\overline{MRDC}$  と  $\overline{MWTC}$  の両方を High にし、 $T_c$  の最初に  $\overline{MRDC}$ 、 $\overline{MWTC}$  のどちら

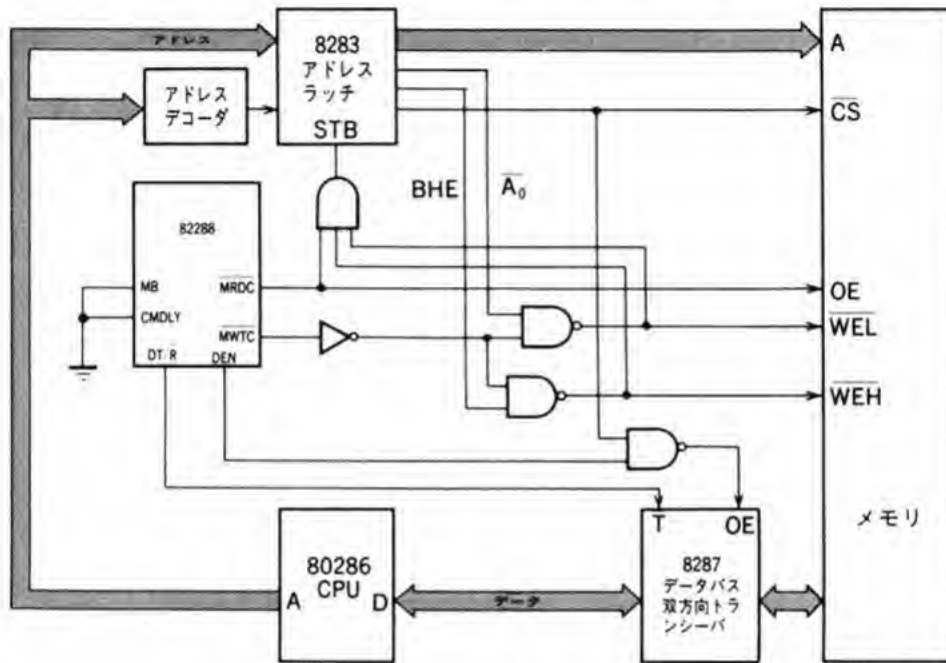


図 8・14 特別ストロブ信号を使用したメモリ構成  
(iAPX 286 Hardware Reference Manual  
(Order No.210760-001), 4-8, Fig 4-2 より引用)

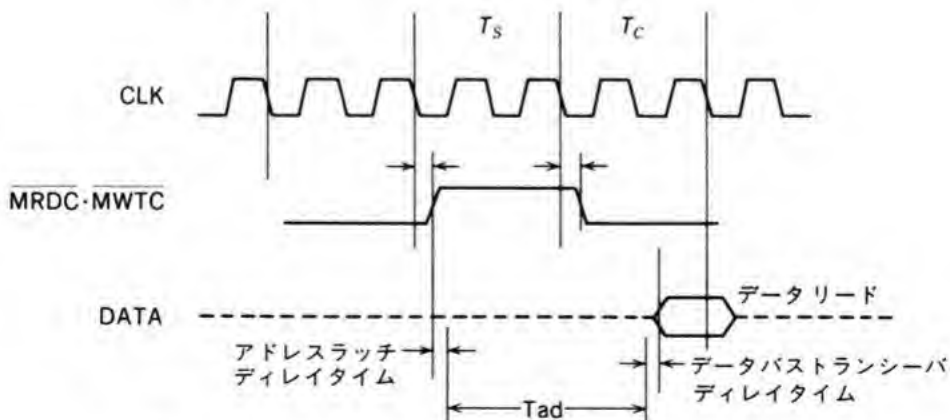


図 8・15  $\overline{MRDC} \cdot \overline{MWTC}$  をアドレスラッチのストロブ信号として使用した場合のメモリ応答最大余裕時間

## 8 80286 のハードウェア

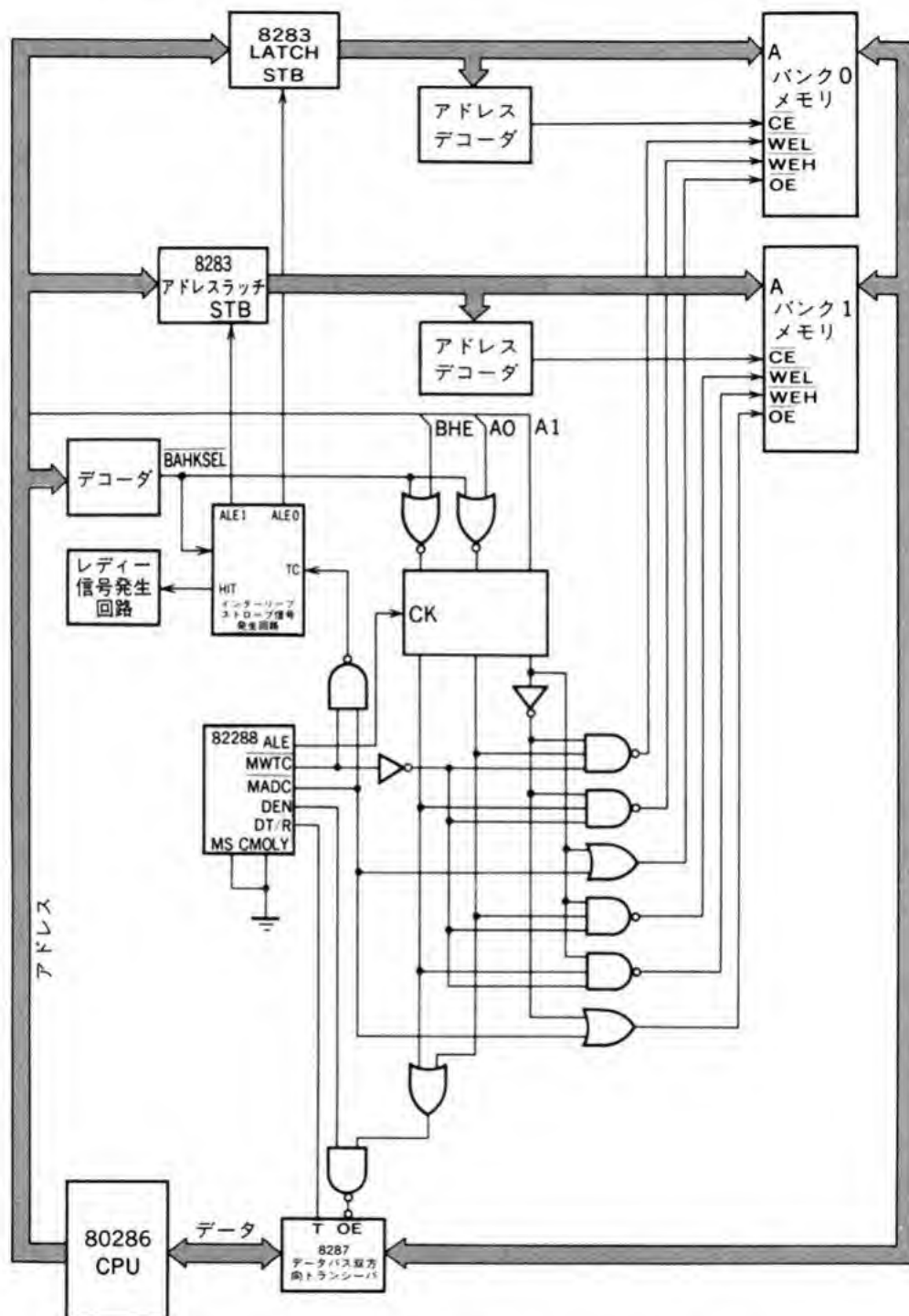


図 8・16 パイプラインアクセスを利用したメモリ構成  
(iAPX 286 Hardware Reference Manual  
(Order No.210760-001), 4-10, Fig 4-3 より引用)

か一方を Low にするから  $\overline{\text{MRDC}}$  と  $\overline{\text{MWTC}}$  の論理 AND をとることによって図 8・15 に示すような信号が作られる。

この構成では、コマンド信号の立ち上がりでアドレスラッチにストロープ信号が供給され、アドレスバスに新しいアドレスが出力される。したがって、図 8・12 に示した基本的メモリ構成より 1 CLK だけ大きなメモリ応答の最大余裕時間  $T_{ad}$  が得られる。

〔3〕 **パイプラインアクセスの利用** 80286 がもつパイプライン的なアドレス出力を有効に利用するために、図 8・16 に示すようなメモリ構成にすることができる。この構成では、メモリをアドレスバスの  $A_1$  の値によって、2つのメモリバンクに分割し、それぞれのメモリバンクには独立したアドレスバスからアドレスを供給する。 $A_1$  の値が 0, 1, 0, 1, … となるように、80286 がバスサイクルを実行するならば、バンク 0 のメモリがデータの入出力を実行しているときに、次のバスサイクルのアドレスをバンク 1 に送ることができる。また、バンク 1 のメモリの実行終了を待たずに、さらに次のアドレスをバンク 0 に供給することができる。もちろん、80286 は同じバンクに連続してバスサイクルを実行する場合もある。このときはパイプライン的なメモリ参照はできず、バスサイクルにウェイトステートを挿入する必要がある。しかし、プログラムの平均的な実行状況において、同じメモリバンクへの連続的な参照の回数は、全体のメモリ参照の 7% 程度である。したがって、図 8・16 に示したようなメモリ構成の効果は十分にある。

### 8-3 I/O インタフェース

I/O インタフェースは CPU と周辺装置の間で、データ変換を行う回路である。コンピュータと周辺装置との間のデータ伝送には、セントロニクス、RS-232-C などのいくつかの標準的な方法がよく使用される。また、これらの方法でデータの入出力を行う I/O インタフェースも、半導体各社で製造されている。

図 8・17 に I/O インタフェースの特徴を表す簡単なモデルと、80286 のバスとの接続を表す。I/O インタフェースは内部に、データを一時的に保存するためのレジスタまたは I/O インタフェース自身の動作モードを決めるためのレジスタなどをもち、これらのレジスタはアドレスデコーダの設計によって、特定の I/O アドレスが与えられている。80286 では、IN 命令、OUT 命令のオペランドに I/O アドレスを指定することによって、I/O インタフェース内部の各レジスタからデータを入力したり、または逆にデータを書いたりすることができる。このとき実行されるバスサイクルは、メモリのリード、ライトで実行されるものと、基本的には同じである。80286 は  $A_0$ - $A_{15}$  に I/O アドレスを出力し、また  $\overline{S_0}$ ,  $\overline{S_1}$ ,  $M/\overline{IO}$

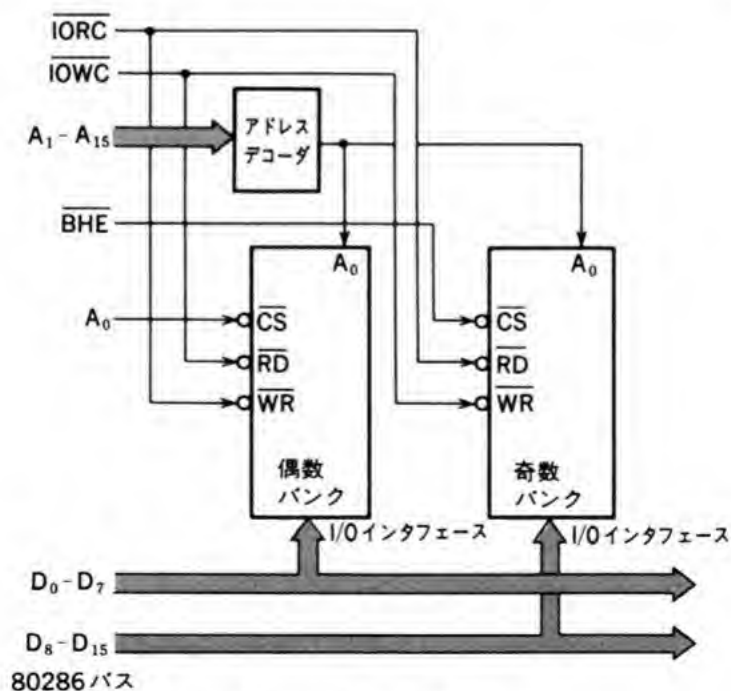


図 8・17 I/O インタフェース

### 8-3 I/O インタフェース

に I/O のリードまたはライトであることを示すステータス信号を出力する。I/O アクセスのとき、 $A_{16}-A_{23}$  は 0 になっている。

たとえば、図 8-17 に示す回路例では I/O インタフェース内部に 2 種類の 8 ビットレジスタがあり、I/O インタフェースの  $A_0$  端子を 0 にするか、1 にするかによって、どちらかのレジスタが選択されるものとする。

I/O インタフェースとバスの接続も、メモリを接続する場合と基本的には変わらない。ただし、IN 命令、OUT 命令によって、リードまたはライトが実行できるように、コマンド信号として  $\overline{IORC}$ 、 $\overline{IOWC}$  に応答するように接続する。その他はメモリの接続と同様に、データバスの下位バイトに接続する I/O インタフェースの  $\overline{CS}$  (チップセレクト) 端子には、アドレスバスの  $A_0$  を接続し、データバスの上位バイトに接続する I/O インタフェースの  $\overline{CS}$  端子には、 $\overline{BHE}$  を接続する。したがって、データバスの下位バイトに接続する I/O インタフェースの内部レジスタはすべて偶数アドレスをもち、逆にデータバスの上位バイトに接続した I/O インタフェースの内部レジスタはすべて奇数アドレスをもつ。

このように、80286 のバスと I/O インタフェースの接続は、コマンド信号の種類が異なる他は、メモリの接続の場合と変わるところはない。したがって、I/O インタフェースにも、 $\overline{MRDC}$ 、 $\overline{MWTC}$  のコマンド信号と、 $A_0-A_{23}$  までのアドレスをデコードした信号を供給してもかまわない。このとき、I/O インタフェース内部のレジスタのリード、ライトは、IN 命令または OUT 命令ではなく、MOV 命令などを直接使用することができる。I/O インタフェースをメモリと見たてて、バスに接続することを**メモリマップト I/O** (メモリに配置された I/O) と呼ぶ。

## 8-4 ローカルバス制御

図8・18にローカルバスの構成を示す。ローカルバスは一般に1つのCPUによって制御され、他のCPUとの間で共有されないようなバスである。しかし、ローカルバスにも**DMAコントローラ**のようなバスを能動的に制御するCPU以外のバスマスタをもつ場合がある。DMAコントローラは、メモリ-メモリ間、またはメモリ-I/O間のデータ転送を行うコントローラであり、大量のデータを連続的に転送するとき有効である。

DMAコントローラはCPUから送られるコマンドまたは他の回路から送られるリクエスト信号によって動作を開始する。このとき、CPUから一時的にバスを預かるような制御をしなければ、DMAコントローラとCPUとの間で信号の衝突が発生する。80286の場合、**HOLD信号**と**HLDA信号**によって、ローカルバスの制御を実行する。たとえば、DMAコントローラがHOLD信号をHighにしたとき、80286は $D_0-D_{15}$ 、 $A_0-A_{23}$ 、 $\overline{BHE}$ 、 $\overline{S_0}$ 、 $\overline{S_1}$ 、 $M/\overline{IO}$ 、 $\overline{COD}/\overline{INTA}$ 、 $\overline{LOCK}$ 、 $\overline{PEREQ}$ などの、バス制御に関係する端子をハイインピーダンス状態にしてから、HLDA端子にHighの信号を出力する。この後、DMAコントローラが80286に代わってバスを制御する。

もちろん、DMAコントローラが一連の処理を実行し終ったとき、HOLD信号をLowに戻す。このとき、80286はHLDA信号をLowに戻してから、再びバス制御を取り返すことができる。

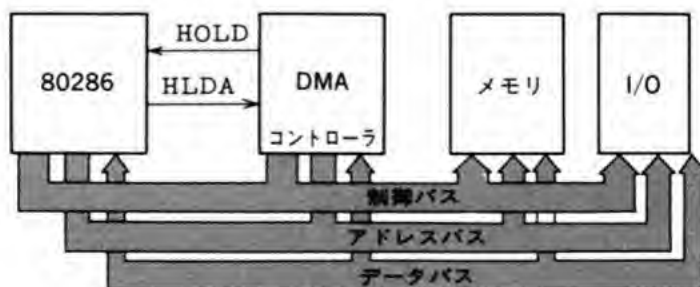
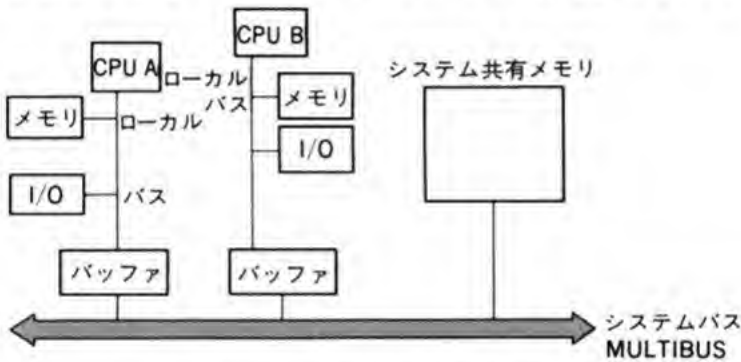


図 8・18 ローカルバス

## 8-5 システムバス制御

コンピュータの負荷が大きくなった場合、複数の CPU に負荷を分散させることによって、問題を解決することができる。たとえば、本来の計算処理を実行する CPU と、入出力処理を専門に実行する CPU に分割する。このようなとき、システムバスまたは標準バスと呼ばれる、CPU の特性には関係ない標準のバスを用意することが有効である。

システムバスの構成は図 8・19 に示すようになる。システムバスには、複数の CPU を接続することができ、それぞれの CPU はローカルバスを使用して、独立に動作することができる。また、システムバスに接続されるメモリは、システ



注) MULTIBUS はインテル標準のシステムバスである。

図 8・19 システムバス

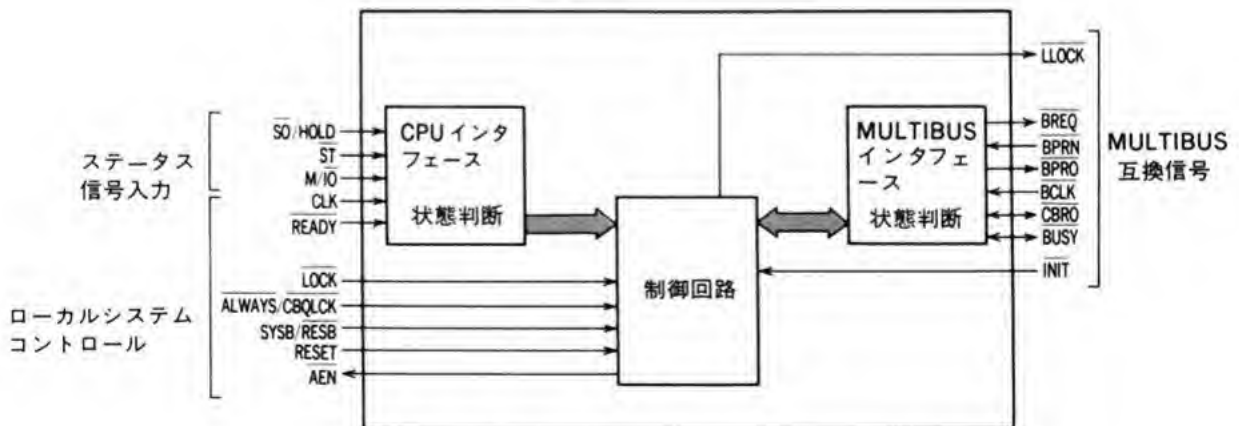


図 8・20 82289 バスアービタ

(Microsystem Components Handbook Microprocessors Vol.1)  
(Order No. 230843-003), 4-167, Fig 1 より引用

## 8 80286 のハードウェア

ムバス上の任意の CPU から共有される共有メモリとして働く。たとえば、CPU A によって処理したデータを CPU B において、さらに別の処理を実行するためデータを渡したいようなとき、CPU A が共有メモリにデータを書いた後で、CPU B が共有メモリからデータを読むという方法で実現できる。

このように、システムバスは複数の CPU から共有されるものであるから、システムバス上で信号の衝突が発生しないような柔軟な制御が必要である。システムバスの制御を行う回路を**バスアービタ**と呼ぶ。80286 のバスアービタには **82289** を使用することができる。82289 のブロック図を図 8・20 に示す。

バスアービタは、システムバスに接続される CPU ボード上に 1 個ずつ配置し、バスアービタ相互の接続によって決まる優先順位に従って、システムバスを要求する CPU の中で、最も優先順位の高いものにだけシステムバスの使用を許す。システムバスを獲得できなかったバスアービタは、システムバスへのバスバッファ出力をハイインピーダンス状態にし、CPU はウェイト状態になるように 82284 を制御する。

# 9.

## 数値演算コプロセッサ80287

浮動小数点演算を高速で実行することは、マイクロコンピュータにとっても重要な働きである。しかし、そのような機能を必要としない場合もあるから、浮動小数点演算機能をCPUに内蔵することは不経済である。そこで、8086のときに**コプロセッサ**という概念が生まれた。8086だけを使用すれば、ワードタイプまでの整数演算しかできないが、コプロセッサ8087を追加することによって、浮動小数点演算も実行できるようになる。コプロセッサは単独では動作することができない。80286の場合は、数値演算コプロセッサ80287をもつ。80287の内部は8087と同様である。ただし、80287は、ホストとなるCPU(80286または80386)との接続方式が8087と8086の接続の場合と異なる。したがって、8087の応用プログラムは、80287においてもほとんど変更することなく動作する。

## 9-1 80287 のアーキテクチャ

図 9・1 に 80287 のレジスタ構成を示す。レジスタスタックと呼ばれる 80 ビットのレジスタが 8 本ある。レジスタスタックは、演算の対象となる浮動小数点データを格納するレジスタであるが、各レジスタを識別するためのレジスタ名はもたず、メモリのアドレスのように 0 から 7 までのアドレスが与えられている。さらに、これらのレジスタスタックは、ちょうどメモリのスタックのように使用される。図に示すステータスレジスタ中の 3 ビットの TOP フィールドの値によって、現在のスタックトップのレジスタを表す。スタックトップのレジスタがアキュムレータとなる。このように、80287 はいわゆるスタックマシンとして使用するこ

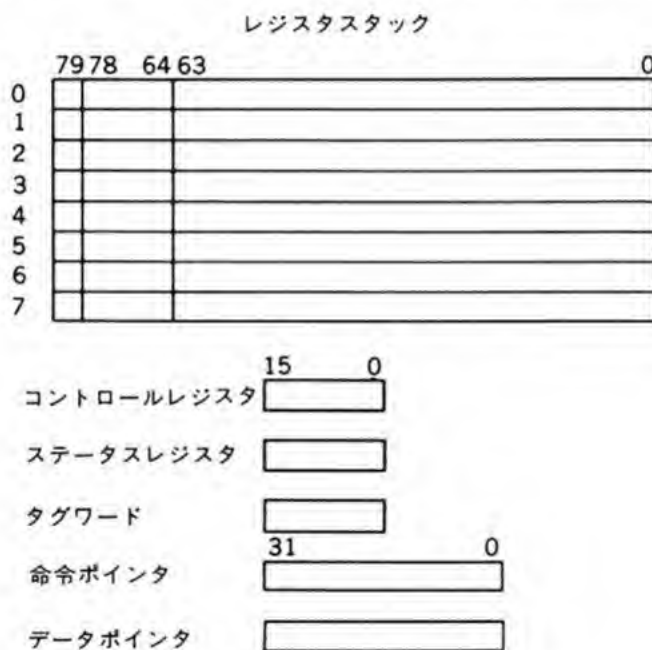
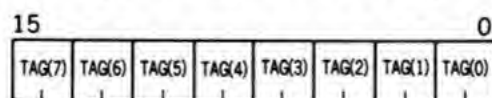


図 9・1 80287 のレジスタ構成



タグの値

- 00B ⇒ 正しい数値データをもつ。  
 01B ⇒ 数値データ 0 をもつ。  
 10B ⇒ 不正データまたは無限大をもつ。  
 11B ⇒ あき状態である。

図 9・2 タグワード

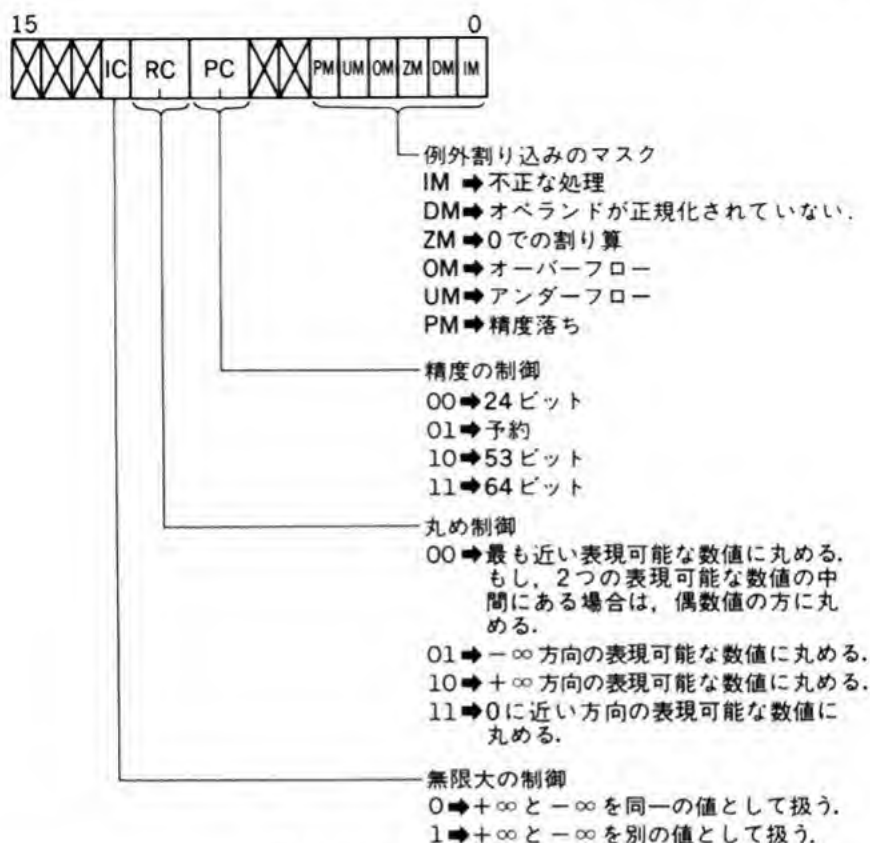


図 9・3 コントロールレジスタ

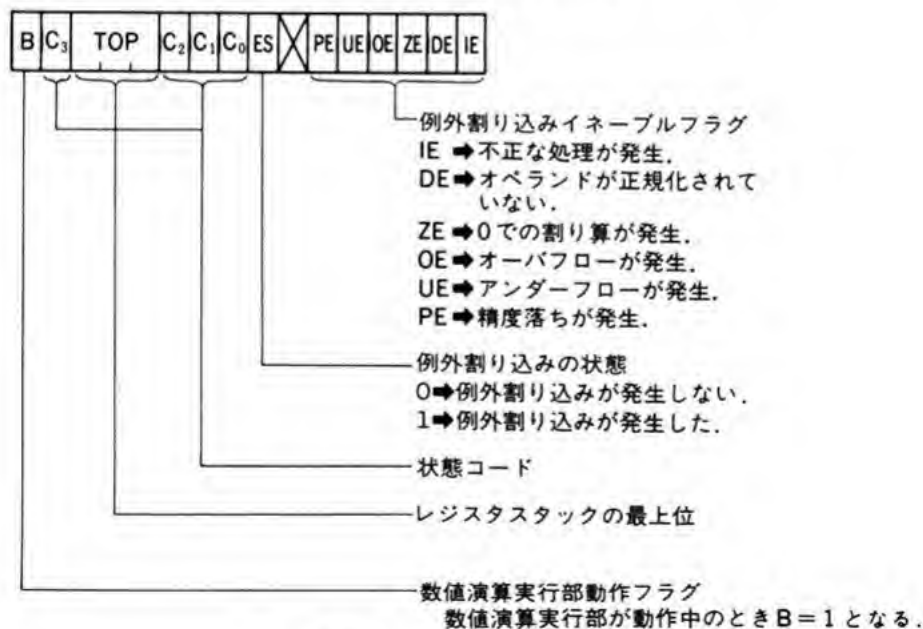


図 9・4 ステータスレジスタ

## 9 数値演算コプロセッサ 80287

とができる。

タグワードは、図9・2に示すように16ビットのレジスタで、2ビットの各フィールドがそれぞれレジスタスタックに対応し、レジスタスタックの状態を表す。タグワードは、ほとんど80287の内部動作において、レジスタスタックの値の評価を高速化するために使用されるが、ユーザプログラムから直接にタグの値を読んだり、変更することも可能である。

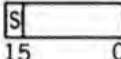
コントロールレジスタは、80287の動作を制御するレジスタである。コントロールレジスタの構成を図9・3に示す。ビット0からビット5までは数値演算の実行における例外割り込みのマスクである。これらのマスクビットを1にしておけば、図9・3に示す状況が発生しても、80287は80286への割り込み要求信号を発生しない。PCの2ビットはレジスタスタックの演算結果の精度を制御する。PCの値によって、仮数部の有効桁を24ビット、53ビット、64ビットのどれかに決めることができる。また、ICの値によって無限大数の扱いを制御することもできる。すなわち、IC=0のとき $+\infty$ と $-\infty$ は同じ値として扱うのに対して、IC=1のとき $+\infty$ と $-\infty$ を別の値として扱う。なお、80287は80ビットの特別なコードで、 $+\infty$ および $-\infty$ を表現することができる。


ステータスレジスタは80287の処理状況を表す、80286のFLAGに相当するレジスタである。図9・4にステータスレジスタの構成を示す。ビット0からビット5は、数値演算処理において例外割り込みが発生したときに1になる。各ビットの例外割り込みの種類は、コントロールレジスタのビット0からビット5までのマスクビットに対応している。80287の例外処理に対しては、コントロールレジスタでマスクすることによって、割り込み要求信号を発生させずに、ステータスレジスタを読み取って、80287の状態を知ることができる。ESはマスクされていない例外割り込みが発生したときに1になる。 $C_0$ 、 $C_1$ 、 $C_2$ 、 $C_3$ はデータの比較命令の結果などが残る。TOPの3ビットはレジスタスタックの最上位のレジスタを指定する。Bビットは80287が実行中のとき1になる。


## 9-2 データの表現

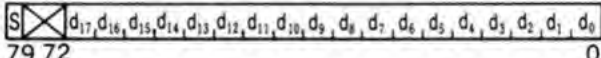
80287 は図 9・5 に示すようなデータを表すことができる。すなわち、メモリに定義された図に示す各データをレジスタスタックに転送することができる。このとき、すべてのデータは、**テンポラリリアル**と呼ばれる 80 ビットの太きさの浮動小数点表現に自動的に変換されてから、スタックレジスタに代入される。したがって、スタックレジスタ上では、データはすべてテンポラリリアルの形で表現される。逆に、スタックレジスタの値をメモリに記録するとき、図に示す任意の形式に変換することができる。

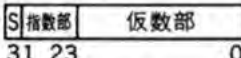
整数は最上位ビットが符号ビットとなり、マイナスの値は 2 の補数で表される。大きさは 16 ビット、32 ビット、64 ビットの整数がある。また、パケットデシマル

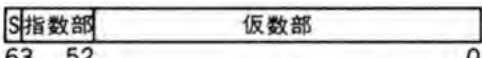
ワードインテジャ  
(16 ビット整数)  (2 の補数表現)

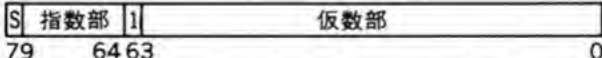
ショートインテジャ  
(32 ビット整数)  (2 の補数表現)

ロングインテジャ  
(64 ビット整数)  (2 の補数表現)

パケットデシマル 

ショートリアル  
(32 ビット浮動  
小数点表現)  (ビット 22 とビット 23 の間に 1. を仮定する)

ログリアル  
(64 ビット浮動  
小数点表現)  (ビット 51 とビット 52 の間に 1. を仮定する)

テンポラリリアル  
(80 ビット浮動  
小数点表現)  (ビット 62 とビット 63 の間に 小数点を仮定する)

ここで

S 符号ビット  $\begin{cases} S=0 & \text{プラス} \\ S=1 & \text{マイナス} \end{cases}$

$d_i (i=0 \sim 17)$  4 ビットで 1 桁の 10 進数を表す。

ショートリアルは  $S1. \text{仮数部} \times 2^{(\text{指数部} - 127)}$  を表す。

ログリアルは  $S1. \text{仮数部} \times 2^{(\text{指数部} - 1023)}$  を表す。

テンポラリリアルは  $S1. \text{仮数部} \times 2^{(\text{指数部} - 16383)}$  を表す。

図 9・5 80287 のデータ

## 9 数値演算コプロセッサ 80287

は特別な形式で表す整数で、10進数の1桁を4ビットで表す。80ビットの下位72ビットで17桁までの10進整数を表すことができる。最上位ビットは符号ビットとなる。

浮動小数点データには、大きさが32ビット、64ビット、80ビットの3種類のものがある。スタックレジスタ中のデータは、すべて80ビットのテンポラリリアルで表現される。メモリからスタックレジスタにデータをリードするとき、またはスタックレジスタの値をメモリにライトするとき、80287はテンポラリリアルと他のデータタイプの間のデータ変換を自動的に実行する。

### IEEE 754 と 80287

コンピュータの互換性の問題の中に、浮動小数点データのフォーマットの問題がある。同じFORTRANのプログラムを実行しても、コンピュータの機種によって浮動小数点データの表現が異なり、丸め誤差の影響によって出力データまたはその動作に違いが発生する場合がある。そこで、浮動小数点データのフォーマットを標準化するためにIEEE 754 浮動小数点データ標準化案がある。8087と同様に、80287もIEEE 754に準じた単精度および倍精度浮動小数点データを扱う。

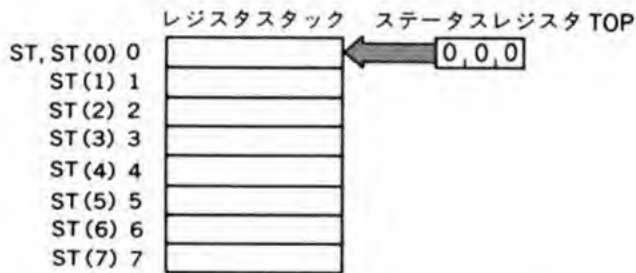
しかし8087、80287内部では、データはすべて80ビットの大きさのテンポラリリアルと呼ばれる8087、80287の内部表現に変換されて扱われる。32ビット長の単精度または64ビット長の倍精度浮動小数点データをテンポラリリアルに拡張することは、演算途中のオーバフローに対するマージンとなる。たとえば、簡単のために10進2桁までのデータを扱うことができるコンピュータがあるとして、 $10 \times 10 \div 2$ の演算を実行すると結果は50であるが、途中で100というデータが発生してこのコンピュータの扱えるデータの大きさを超えてしまう。演算順序を変更するなどの方法でオーバフローを避けることもできるが、最も簡単な方法はこのコンピュータを10進3桁までのデータを扱えるようにすることである。

また、8087、80287は整数とBCDを扱うこともできるが、これらのデータも8087、80287内部ではテンポラリリアルに変換して扱われることに注意する。

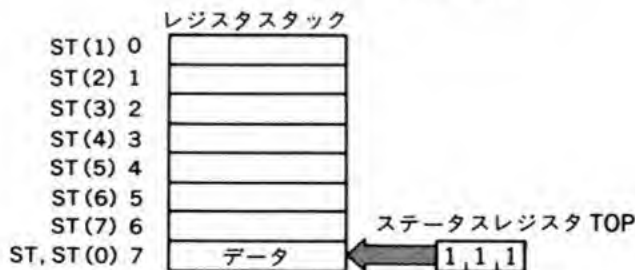
## 9-3 レジスタスタックの基本的使用

〔1〕 レジスタスタックの構成 レジスタスタックは、図9・6 (a) に示すように 80 ビットの 8 本のレジスタで、0 から 7 までの番地が与えられる。ステータスレジスタ TOP の 3 ビットの値が、スタックレジスタの現在の最上位のレジスタを表す。最上位のスタックレジスタは、アキュムレータとして 80287 の演算処理の中心となるデータを保存する。最上位のスタックレジスタを 80287 の命令のオペランドに指定するときには、ST と書く。80287 は、純粹のスタックマシンではなく、スタックレジスタの最上位以外のレジスタを任意に参照することもできる。このとき、ST (1) から ST (7) の名前で、それぞれのスタックレジスタを指定することができる。もし、ST が 0 番のスタックレジスタであれば、1 番のスタックレジスタは ST (1)、また 7 番のスタックレジスタは ST (7) となる。また、図9・6 (b) に示すように、7 番のスタックレジスタが ST であれば、0 番のスタックレジスタは ST (1) となり、6 番のスタックレジスタが ST (7) となる。

〔2〕 レジスタスタックへの代入 レジスタスタックに値を代入するために



(a) レジスタスタックの初期状態



(b) データの入った後のレジスタスタック

図 9・6 レジスタスタックの使用

表 9・1 FLD 命令の種類

オペコード ニーモニック	オペランド	動作
FLD	DWORD タイプメモリ	メモリに定義したショートリアルデータをテンポラリアルに変換してスタックレジスタに PUSH する。
	QWORD タイプメモリ	メモリに定義したロングリアルデータをテンポラリアルに変換してスタックレジスタに PUSH する。
	TBYTE タイプメモリ	メモリに定義したテンポラリアルデータをスタックレジスタに PUSH する。
	ST (i) (ここで i=0~7)	スタックレジスタ ST (i) の値をスタックレジスタの最上位に PUSH する。
FILD	WORD タイプメモリ	メモリに定義したワード整数データをテンポラリアルに変換して、スタックレジスタに PUSH する。
	DWORD タイプメモリ	メモリに定義したショート整数データをテンポラリアルに変換してスタックレジスタに PUSH する。
	QWORD タイプメモリ	メモリに定義したロング整数データをテンポラリアルに変換してスタックレジスタに PUSH する。
FBLD	TBYTE タイプメモリ	メモリに定義したバクトデシマルデータをテンポラリアルに変換してスタックレジスタに PUSH する。

表 9・2 FST 命令

オペコード ニーモニック	オペランド	動作
FST ESTP	DWORD タイプメモリ	レジスタスタックの最上位にあるデータをショートリアルに変換して、メモリに保存する。
	QWORD タイプメモリ	レジスタスタックの最上位にあるデータをロングリアルに変換して、メモリに保存する。
	TBYTE タイプメモリ	レジスタスタックの最上位にあるデータをテンポラリアルのまま、メモリに保存する。
	ST (i) (ここで i=0~7)	レジスタスタックの最上位にあるデータを他のレジスタスタックに保存する。
FIST FISTP	WORD タイプメモリ	レジスタスタックの最上位にあるデータをワード整数に変換して、メモリに保存する。
	DWORD タイプメモリ	レジスタスタックの最上位にあるデータをショート整数に変換して、メモリに保存する。
	QWORD タイプメモリ	レジスタスタックの最上位にあるデータをロング整数に変換して、メモリに保存する。
FBSTP	TBYTE タイプメモリ	レジスタスタックの最上位にあるデータをバクトデシマルに変換して、メモリに保存する。

オペコードニーモニックで最後に P の付いているものは、動作の最後にレジスタスタックの最上位の TAG を 11B にして、TOP の値を 1 だけ増分する操作を含む。

は、FLD 命令を使用する。FLD 命令は

FLD オペランド

のように書き、ステータスレジスタの TOP から 1 だけ減算してから、TOP が指定するレジスタスタックに、オペランドで指示したデータを代入する。このとき、オペランドのデータは自動的にテンポラリリアルに変換される。このように、FLD 命令の動作はメモリへの PUSH 命令とよく似ている。また、FLD 命令には、オペランドで指定するデータのタイプによって、表 9・1 に示す種類がある。

80287 のアセンブリ言語の命令では、ロングリアルとロングインテジャのデータは、どちらも 64 ビットの大きさであるから、オペランドのタイプだけでは区別できない。また、パケットデシマルとテンポラリリアルの場合も、どちらも 80 ビットの大きさであるから、オペランドのタイプだけでは区別できない。そこで、80287 の命令のニーモニックは表に示したように、F の次に I または B を書くことによって、リアルタイプ、インテジャタイプ、パケットデシマルのデータを区別する。図 9・6 (a) に示したように、TOP の値が 0 のとき、FLD 命令を使用して、メモリまたは他のレジスタスタックから、レジスタスタックの最上位にデータを代入すれば図 (b) に示すようになる。すなわち、TOP の値は、3 ビットの演算でキャリーを無視するから 0 から 1 を減算することによって 7 になり、7 番のレジスタスタックが新しい ST になる。

このように、FLD 命令によって、レジスタスタックへのデータの PUSH を繰り返し実行したとき、レジスタスタックに代入できるデータの数は最大 8 までである。レジスタスタックにデータを代入するためには、対応する TAG の値がそのレジスタスタックがデータをもっていないことを示す 11B の値でなければならない。もし、すべてのレジスタスタックがデータをもつときに、FLD 命令を実行してもデータを代入することはできない。

〔3〕 **レジスタスタックのストア** FLD 命令とは逆に、レジスタスタックの最上位のデータを、メモリまたは他のレジスタスタックに書くためには FST 命令を使用する。FST 命令は

FST オペランド

のように書く。FST 命令の場合も、扱うデータタイプによって表 9・2 に示す種類がある。FST 命令を実行したとき、レジスタスタックの最上位のデータが、命令のニーモニックとオペランドタイプの組み合わせによって決まるデータに変換

## 9 数値演算コプロセッサ 80287

されてから、メモリまたは他のレジスタスタックに書かれる。

FST 命令の場合は、オペコードのニーモニックの最後に P が付いているものと、付いていないものがある。最後に P が付いている FST 命令は POP 操作を含む。すなわち、命令の最後でレジスタスタックの最上位の TAG を 11B に戻してから、TOP の値に 1 を加算する。したがって、図 9・6 (b) の状態で FSTP 命令を実行したとき、レジスタスタックの状態は図 (a) に示すようになる。FST 命令のニーモニックの最後に P が付いていない命令は上のような POP 操作を含まないから、レジスタスタックの最上位は変化しない。

### スタックマシン

スタックマシンとは CPU 内部のレジスタがメモリのスタックのような構成になっているコンピュータのことをいう。演算処理はレジスタスタックの最上位にある 1 つまたは 2 つのレジスタに対して実行され、一般に命令のオペランドを指定する必要はない。演算結果は最上位のレジスタに残る。

このように、スタックマシンでは、最も新しいデータがレジスタスタックの最上位に位置し、あまり使用されないデータはスタックの底に残っていくことになる。

80287 は基本的にスタックマシンの構成をもっているが、完全に純粋なスタックマシンではない。ST (3) などのレジスタ名を用いて特定のレジスタをオペランドに明示する命令も備えている。ただし、ST (i) というレジスタ名はスタックトップの位置によって指定されるレジスタが相対的に決まることに注意する必要がある。

## 9-4 演算命令と関数命令

〔1〕 基本演算命令 80287 は、加減乗除の基本的な算術演算命令と各種の関数命令をもつ。表 9・3 に基本的な算術演算命令を示す。算術演算命令では、オペランドをもたないもの、メモリオペランドを 1 つだけ指定するもの、そして 2 つのレジスタスタックをオペランドに指定するものがある。しかし、すべての演

表 9・3 算術演算命令

オペコードニーモニック		オペランド	動作
加算	FADD	オペランドなし メモリ ST, ST(i) ST(i), ST	FADDP ST(1), ST と同じ $ST \leftarrow ST + \text{メモリ}$ $ST \leftarrow ST + ST(i)$ $ST(i) \leftarrow ST(i) + ST$
	FIADD	メモリ	$ST \leftarrow ST + \text{メモリ}$
	FADDP	ST(i), ST	$ST(i) \leftarrow ST(i) + ST$ , レジスタスタックの POP
減算	FSUB	FSUBR メモリ ST, ST(i) ST(i), ST	FSUB は FSUBP ST(1), ST と同じ FSUBR は FSUBRP ST(1), ST と同じ $ST \leftarrow ST - \text{メモリ}$ $ST \leftarrow ST - ST(i)$ $ST(i) \leftarrow ST(i) - ST$
	FISUB	FISUBR メモリ	$ST \leftarrow ST - \text{メモリ}$
	FSUBP	FSUBRP ST(i), ST	$ST(i) \leftarrow ST(i) - ST$ , レジスタスタックの POP
乗算	FMUL	オペランドなし メモリ ST, ST(i) ST(i), ST	FMULP ST(1), ST と同じ $ST \leftarrow ST \times \text{メモリ}$ $ST \leftarrow ST \times ST(i)$ $ST(i) \leftarrow ST(i) \times ST$
	FIMUL	メモリ	$ST \leftarrow ST \times \text{メモリ}$
	FMULP	ST(i), ST	$ST(i) \leftarrow ST(i) \times ST$ , レジスタスタックの POP
除算	FDIV	FDIVR メモリ ST, ST(i) ST(i), ST	FDIV は FDIVP ST(1), ST と同じ FDIVR は FDIVRP ST(1), ST と同じ $ST \leftarrow ST / \text{メモリ}$ $ST \leftarrow ST / ST(i)$ $ST(i) \leftarrow ST(i) / ST$
	FIDIV	FIDIVR メモリ	$ST \leftarrow ST / \text{メモリ}$
	FIDIVP	FDIVRP ST(i), ST	$ST(i) \leftarrow ST(i) / ST$ , レジスタスタックの POP

オペコードニーモニックの最後に R のついた命令は逆演算となる。たとえば、FSUBR ST(1), ST の動作は、 $ST(1) \leftarrow ST - ST(1)$  となり、また、FDIVR ST, ST(2) は  $ST \leftarrow ST(2) / ST$  となる。

算命令において、処理の対象の一方は必ず最上位のレジスタスタック ST である。オペランドにメモリを指定した命令は、メモリに定義されたデータをテンポラリリアルに変換した値と、ST の間で演算が行われる。さらにメモリオペランドをもち、オペコードニーモニックの F の次に I が付いている命令は、メモリに定義されたワード整数、またはショート整数をテンポラリリアルに変換してから、ST との間で演算を実行する。また、メモリオペランドをもち、オペコードのニーモニックの F の次に I が付いていない命令は、メモリに定義されたショートリアルまたはロングリアルをテンポラリリアルに変換したデータと ST との間で演算を実行する。演算命令のメモリオペランドに、テンポラリリアル、パケットデシマル、ロング整数を指定することはできない。

減算と除算のオペコードニーモニックの最後に R が付いている命令は逆演算命令となる。たとえば

FSUB ST,ST(2)

の命令は、レジスタスタックの最上位 ST から ST (2) を減算した結果を ST に代入する。これに対して

FSUBR ST,ST(2)

の命令は、ST (2) から ST を減算した結果を ST に代入する。このように逆演算命令は、被減数と減数とが逆になる。除算の場合も同様に、オペコードニーモニックの最後に R が付いた命令は逆演算になり、被除数と除数が逆になる。

演算命令においても、オペコードニーモニックの最後に P を付けた命令があり、これは命令の最後にレジスタスタックの POP を実行する。POP 付きの演算命令のオペランドは、必ず両方のオペランドがレジスタスタックであり、右側のオペランドは ST を指定する。

演算命令の中に、オペランドをもたない命令がある。この命令は典型的なスタックマシンの命令であり、演算は必ず ST と ST (1) のレジスタスタックの最上位にある 2 つのレジスタに対して実行され、演算結果を ST (1) に代入してから、最後にレジスタスタックの POP を 1 回だけ実行する。すなわち、レジスタスタックの最上位の 2 つのレジスタが、それらの演算結果をもつ 1 つの最上位のレジスタスタックに置き換わることになる。

〔2〕関数命令 80287 は、加減乗除の四則演算命令の他に表 9・4、表 9・5 に示す関数命令をもつ。関数命令はすべてオペランドをもたない。処理の対象は

表 9・4 80287 基本関数

オペコード ニーモニック	動 作
FSQRT	$ST \leftarrow \sqrt{ST}$
FSCALT	$ST \leftarrow ST \cdot 2^{ST(1)}$
FPREM	$ST \leftarrow ST - ST(1)$ 上の演算の結果 もし、 $ST < ST(1)$ であれば $C_2 = 0$ また、 $ST \geq ST(1)$ であれば $C_2 = 1$
FRNDINT	ST の値を整数に丸める。このとき、小数点以下の数値の扱いはコントロールレジスタの RC によって制御される。
FXTRACT	ST の値を指数部と仮数部に分割し、指数部を ST に代入し、仮数部の値をレジスタスタックにさらに PUSH する。
FABS	$ST \leftarrow  ST $
FCHS	ST の符号を反転させる。

表 9・5 80287 超越関数

オペコード ニーモニック	動 作
FPTAN	ST に定義した $\theta$ に対して、 $Y/X = \tan \theta$ となるような $X, Y$ を求め、 $Y$ を ST に代入してから、 $X$ をレジスタスタックに PUSH する。したがって、実行後 $X$ が ST に、 $Y$ が $ST(1)$ に残る。ただし、 $\theta$ は $0 \leq \theta \leq \pi/4$ でなければならない。
FPATAN	ST に定義した値 $X$ 、 $ST(1)$ に定義した値 $Y$ に対して、 $\theta = \arctan(Y/X)$ となるような $\theta$ を求め、 $ST(1)$ に代入してから、レジスタスタックを POP する。したがって、ST に $\theta$ が残る。ただし、 $0 \leq Y < X < \infty$ でなければならない。
F2XM1	ST に定義した $X$ に対して、 $Y = 2^X - 1$ となるような $Y$ を求め、ST に代入する。ただし、 $0 \leq X \leq 0.5$ でなければならない。
FYL2X	ST に定義した $X$ 、 $ST(1)$ に定義した $Y$ に対して、 $Z = Y \cdot \log_2 X$ となる $Z$ を求め、 $ST(1)$ に代入してから、レジスタスタックを POP する。ただし、 $0 < X < \infty$ かつ $-\infty < Y < +\infty$ でなければならない。
FYL2XP1	ST に定義した $X$ 、 $ST(1)$ に定義した $Y$ に対して、 $Z = Y \cdot \log_2(X+1)$ のような $Z$ を求め、 $ST(1)$ に代入してから、レジスタスタックを POP する。ただし、 $0 \leq  X  < (1 - \sqrt{2}/2)$ かつ $-\infty < Y < +\infty$ でなければならない。

暗黙的に ST または ST(1) に決められている。表 9・4 には基本的な関数命令を示す。この中で、FPREM 命令は ST に定義した値を、ST(1) の値で割ったときの正確な余りを求めるために使用する。ただし、FPREM 命令は 1 回の減算を実行するだけであるから、余りを求めるためにはステータスレジスタの  $C_2$  が 0 になるまで FPREM 命令を繰り返し実行する必要がある。表 9・5 には三角関数、

表 9・6 80287 定数定義命令

オペコード ニーモニック	動 作
FLDZ	レジスタスタックに $+0.0$ を PUSH する。
FLD1	レジスタスタックに $+1.0$ を PUSH する。
FLDPI	レジスタスタックに円周率 $\pi$ を PUSH する。
FLD2T	レジスタスタックに $\text{LOG}_2 10$ を PUSH する。
FLDL2E	レジスタスタックに $\text{LOG}_2 e$ を PUSH する。
FLDLG2	レジスタスタックに $\text{LOG}_{10} 2$ を PUSH する。
FLDLN2	レジスタスタックに $\text{LOG}_e 2$ を PUSH する。

指数関数、対数関数を含む超越関数の命令を示す。しかし、80287 がもつ超越関数の命令は、その適用範囲が限定されたものとなっている。

たとえば、三角関数には、 $\tan$  と  $\arctan$  しかなく、また FPTAN 命令を適用できる範囲は、0 から  $\pi/4$  までである。しかし、0 から  $\pi/4$  までの  $\tan$  が求められれば、任意の角度に対する  $\tan$  はプログラムで計算できる。また、FPTAN 命令の結果は、レジスタスタックの最上位の ST, ST(1) に、 $Y/X$  の比の形で求められるから、 $\sin$ ,  $\cos$  を計算するプログラムも簡単に作成することができる。指数関数、対数関数についても、表 9・5 に示した 3 つの命令だけであるが、80287 は表 9・6 に示すような定数をレジスタスタックに PUSH する命令をもっているのので、これらの命令を組み合わせれば、一般的に使用できる指数関数、対数関数のプログラムを作ることができる。

## 9-5 80286 と 80287 の接続

〔1〕 80286 との接続 8087 がホスト CPU として 8086 を必要としたように、80287 はホスト CPU として、80286 または 80386 を使用しなければならない。しかし、80286 と 80287 の間の制御は、8086 と 8087 間の場合とはまったく異なる方式を採用している。図 9・7 に 80286 と 80287 の接続を示す。この図において、80287 は 80286 に対して一種の I/O インタフェースであると考えることができる。表 9・7 に示すように、80287 に対して、0F8H、0FAH、0FCH の 3 つの I/O

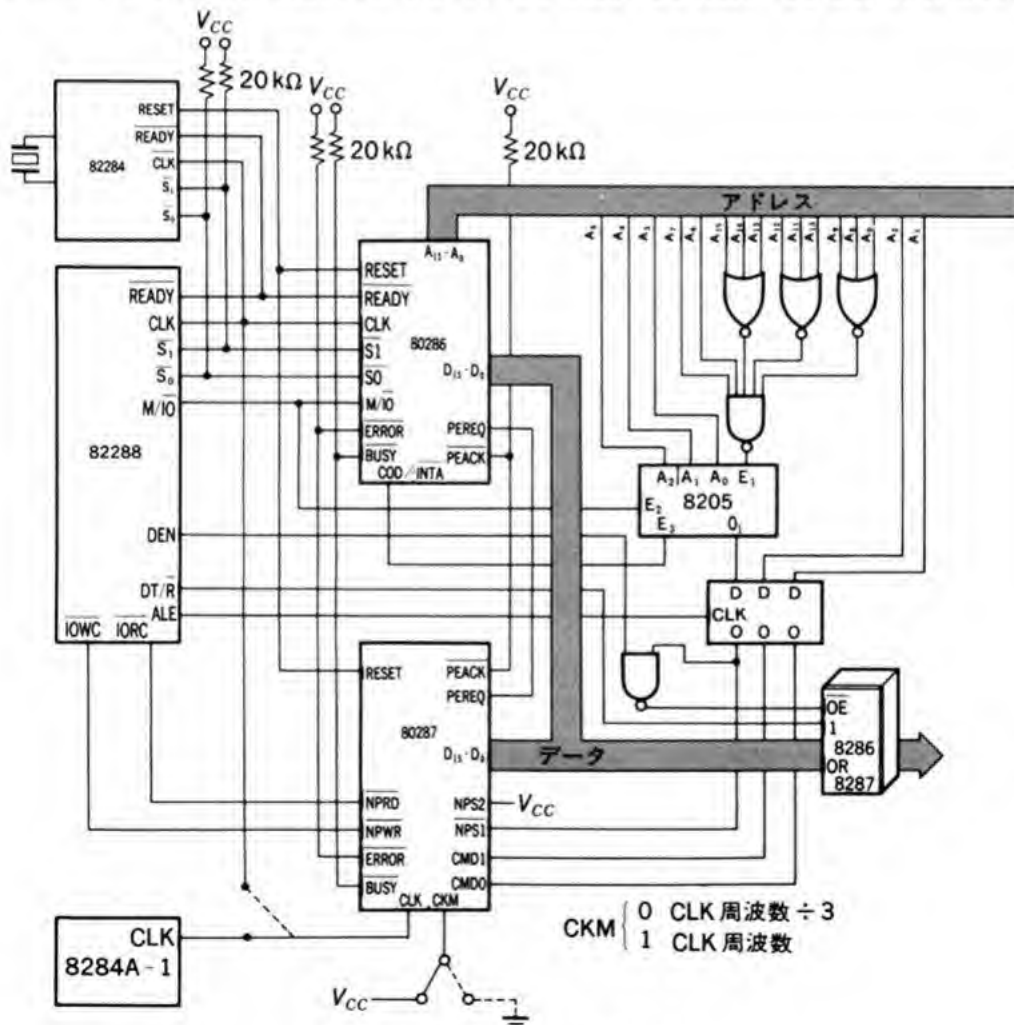


図 9・7 80286 と 80287 の接続

(iAPX286 Programmer's Reference Manual (Order No. 210498-003),  
D-5, Fig4 より引用)

## 9 数値演算コプロセッサ 80287

表 9・7 80287 の I/O アドレス

I/O アドレス	80287 のセレクト信号とコマンド入力			
	NPS 2	NPS 1	CMD 1	CMD 0
00F8H	1	0	0	0
00FAH	1	0	0	1
00FCH	1	0	1	0
00FEH	I/O アドレス 00FEH は、将来のために予約されている。			

アドレスが与えられている。また、80286 は、80287 に対して一種の DMA コントローラであると考えることができる。

8087 の場合は、8086 との間で非常に密な制御を行い、8086 と 8087 のペアを 1 つの CPU のように見ることができた。しかし、8087 に採用された制御方式に欠点がないわけではなかった。それは、8087 は 8086 と同期して動作しなければならない、8086 に供給するものと同一のクロックを 8087 にも供給しなければならない、これは 8086 の製造と 8087 の製造を同じように進めなければならないことを意味する。8087 の場合、8 MHz で動作する 8087 の製造が遅れて、8 MHz の 8086 を使用するユーザも、8087 を接続して使用するときは 5 MHz のクロックで動作させなければならない時期があった。80286 は 80287 を I/O インタフェースとして見ているために、ホスト CPU の動作速度がコプロセッサの動作速度に制限されることはない。80287 が 80286 の I/O の立場になったもう一つの理由は、80286 が非常に厳密なメモリ管理を実行するためである。したがって、80287 は数値演算だけを実行し、80287 の命令におけるメモリ参照も含めて、すべてのメモリ参照を 80286 が実行する。

〔2〕 **80287 のクロック** 80287 の内部回路は、8087 と同様にデューティ比が 33% のクロックで動作する。したがって、82284 が発生するシステムクロックを 80287 の内部回路に供給する場合は、システムクロックを 3 分周しなければならない。このために、80287 は内部に CLK 端子から入力するクロックを 3 分周する分周回路をもつ。80287 の CKM 端子を Low にした場合、分周回路を介して、内部回路にクロックを送ることができる。また、CKM 端子を High にすれば、分周回路を介さずに CLK 端子から入力されたクロックをそのまま内部回路に供給することもできる。図 9・7 に示す例において、82284 が発生する 16 MHz のシステムクロックを 80287 に供給するときは、CKM 端子を Low にしなけれ

ばならない。このとき、80287 の内部回路は 5.3 MHz で動作する。80287 は 80286 とは非同期で動作できるから、システムクロックを 80287 に供給する必要はない。デューティ比 33% のクロックを発生する 8284 A のクロックを、80287 の CLK 端子に供給することも可能である。このとき、CMK 端子を High に接続する。8284 A が 8 MHz のクロックを供給するならば、80287 の内部回路も 8 MHz で動作する。

〔3〕 **80287 の命令の実行** 図 9・7 において、80286 と 80287 の動作について考える。80286 は、80287 の命令 (80287 の命令を総称して **ESC 命令**と呼ぶ) も含めて、すべての命令をフェッチする。もし、フェッチした命令が 80286 の命令であれば 80286 内部で処理し、フェッチした命令が 80287 の命令であれば、80286 はその命令コードを、ちょうど I/O にデータを出力するときのように 80287 に送る。80287 は 80286 から送られた命令をデコードして、実行することができる。80287 での命令の実行は 80286 の動作とは完全に非同期に進む。したがって、80287 が数値演算を実行している間、80286 は次の命令を実行しているかもしれない。

しかし、ここで 1 つの問題がある。それは、80287 が前に送られた命令を実行中であるにもかかわらず、80286 から新しい ESC 命令が送られた場合、前の命令の実行が途中でつぶれてしまう。そこで、80287 は実行中であるかないかを表す  $\overline{\text{BUSY}}$  を 80286 に供給する。80286 は  $\overline{\text{BUSY}}$  が Low であれば、 $\overline{\text{BUSY}}$  が再び High になるまで、ESC 命令のコードを 80287 に送らない。このとき、80286 は命令コードのデコードを一時的に休むことになる (ただし、ESC 命令の中には  $\overline{\text{BUSY}}$  とは無関係に実行されるものもある)。

なお、80286 の  $\overline{\text{BUSY}}$  端子は、80287 とのソフトウェアの同期をとる以外の目的に使用してもかまわない。このために、80286 は WAIT 命令をもつ。80286 が WAIT 命令を実行したとき、 $\overline{\text{BUSY}}$  が Low であれば、 $\overline{\text{BUSY}}$  が再び High になるまで次の命令を実行しない。また、WAIT 命令は、図 9・8 に示すように 80286 の命令が 80287 の命令完了を待たなければならない場合にも使用される。

ESC 命令の中で、FLD 命令または FST 命令のように、メモリからデータをリードしたり、メモリにデータをライトするものがある。このとき、80287 は 80286 を DMA コントローラとして使用する。このときの制御は PEREQ と  $\overline{\text{PEACK}}$  によって実行される。80287 がメモリからデータをリードするときは、80286 が

## 9 数値演算コプロセッサ 80287

RST1 DD ?	⇒4バイトの大きさをもつ変数 RST1 を定義する.
⋮	
FSTP RST1	⇒80287 が ST の値をショートリアル に変換して、変数 RST1 に書く.
WAIT	⇒80287 の FSTP 命令が完全に終了す るまで待つ.
MOV DX,WORD PTR RST1+2	⇒変数 RST1 の上位ワードを DX に代 入する.
MOV AX,WORD PTR RST1	⇒変数 RST1 の下位ワードを AX に代 入する.

図 9・8 WAIT 命令による 80286 と 80287 の同期

メモリからデータをリードしてから、80287 にライトする。また、80287 がメモリにデータをライトするとき、80287 からの要求に従って、80286 が 80287 からリードしたデータをメモリにライトする。

### 8087 の WAIT 命令

8087 は 8086 と同じタイミングで命令をフェッチしてから、また 80287 は 80286 から命令コードを供給されてから命令のデコードを開始し、その処理を実行する。どちらの場合にも現在実行中の命令があるにもかかわらず、次の命令をデコードした場合、前の命令の処理は途中で壊れ結果は残らない。このため、8087 も 80287 も  $\overline{\text{BUSY}}$  端子に実行中か実行中でないかを表す記号を出力し、ホスト CPU において、8086 の場合は 8087 の実行が終了するまで次の命令のデコードを遅延させる処理が必要となる。80286 の場合は 80287 の実行が終了するまで、80287 に命令コードを供給するのを遅延させる必要がある。

このため 8086 の場合は、ESC 命令の前に必ず WAIT 命令を挿入する。しかし 80286 の場合、ESC 命令自体の中に WAIT 命令と同じ機能をもつため、わざわざ WAIT 命令を ESC 命令の前に入れる必要はない。

## 9-6 例 外 処 理

図 9・9 に示すように、80287 は演算処理において 6 種類のエラーを検査する。もし、エラーが発生した場合、80287 はステータスレジスタのビット 0 からビット 5 までのエラーフラグの対応するビットを 1 にする。さらに、コントロールレジスタのビット 0 からビット 5 までの対応するマスクビットが 0 であれば、ステータスレジスタの ES を 1 にして、 $\overline{\text{ERROR}}$  端子に Low の信号を出力する。

8087 の場合は、割り込みコントローラ 8259 A を介して、8086 に割り込み信号を供給したが、80287 の場合は、80287 の  $\overline{\text{ERROR}}$  出力端子を直接 80286 の  $\overline{\text{ERROR}}$  入力端子に接続すればよい。 $\overline{\text{ERROR}}$  が Low のときに、80286 が ESC 命令または WAIT 命令を実行すると、タイプ 16 の割り込みを発生する。

タイプ 16 の割り込み処理で 80287 の例外処理を実行することができる。ステータスレジスタの ES が 1 である間、80287 は  $\overline{\text{ERROR}}$  端子に Low の信号を出力し続けるので、例外処理において FNCLX 命令を実行して、ES を 0 にする必要がある。FNCLX 命令はステータスレジスタのビット 0 からビット 5 までのエラーフラグ、ES、そしてビット 15 の B を 0 にクリアする命令であり、FNCLX 命令自体は  $\overline{\text{ERROR}}$  の状態を無視して実行される。このように、80287 の制御命令の中には  $\overline{\text{ERROR}}$  の状態に関係なく実行されるものがある。

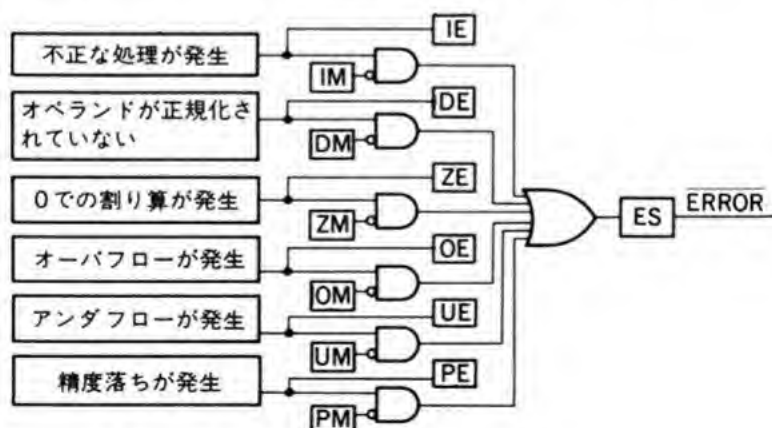


図 9・9 80287 の  $\overline{\text{ERROR}}$  出力

## 9-7 80287 のためのサポート

〔1〕 80286 のサポート 80287 は 80286 から命令コードを与えられて初めて動作する。80287 自体がメモリなどを直接参照することではなく、80287 外部への参照はすべて 80286 が実行する。80287 に対して 80286 が実行する動作は図 9・10 に示すように、MSW の MP, EM, TS の 3 ビットのフラグによって制御することが可能である。

80287 がハードウェアに存在するとき、MP を 1 に設定し、EM を 0 に設定する。逆にハードウェアに 80287 がなく、80287 と同じ機能をもつソフトウェアエミュレータによって 80287 を代用したいときには、MP を 0 に設定し、EM を 1 に設定する。TS については、80286 がタスクスイッチが発生したとき自動的に 1 を設定する。

〔2〕 タスクスイッチにおける 80287 タスクスイッチによって、80286 のレジスタの状態はすべて TSS に自動的に保存されるが、80287 のレジスタについては、プログラムの実行によって保存しなければならない。このとき、TS を利用することができる。すなわち、MP=1, EM=0, TS=1 の条件において ESC 命令または WAIT 命令を実行したいとき、タイプ 7 の割り込みが発生する。TS=1 はタスクスイッチが発生したことを意味するから、いま実行しようとした ESC 命令または WAIT 命令は別のタスクの命令である。したがって、これらの命令を実行する前に、80287 のすべてのレジスタの状態をメモリに保存し、かつ、80287 を初期化してから逆に新しいタスクで使用するレジスタの状態をメモリから 80287

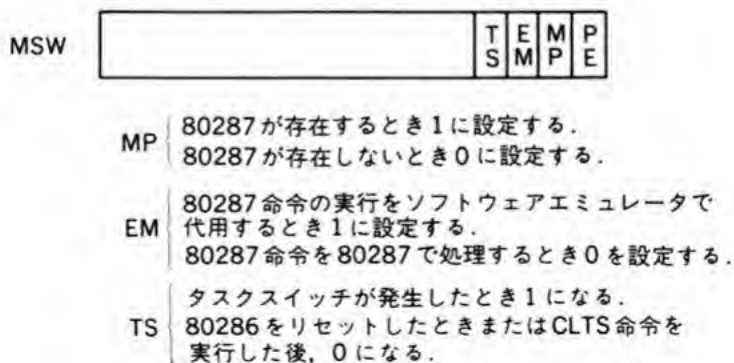


図 9・10 80286 側のサポート

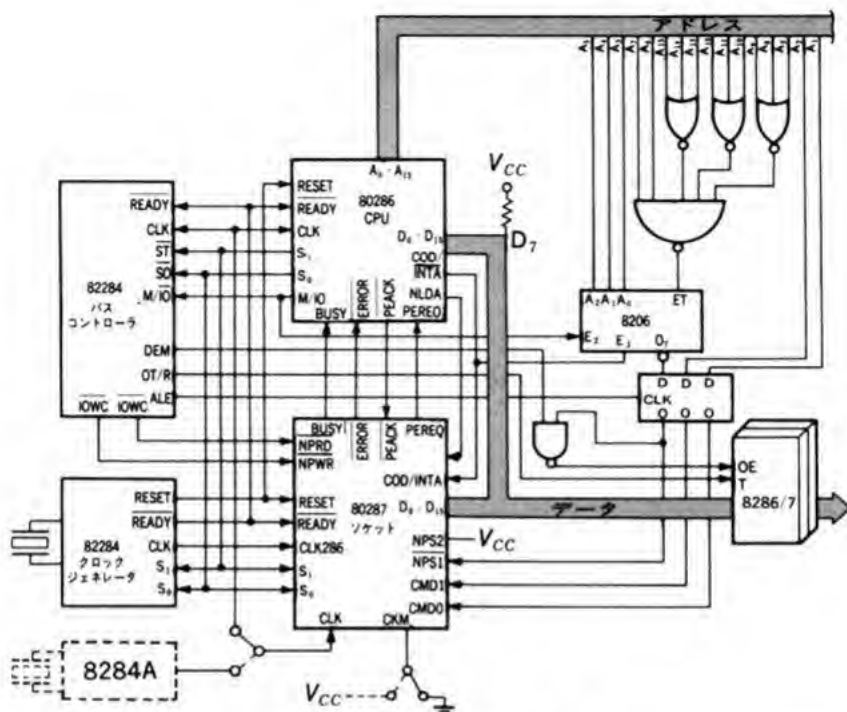


図 9・11 80287 の自動検査

(iAPX286 Hardware Reference Manual (Order No.210760-001),  
6-2, Fig 6-1 より引用)

FND_287:FNINIT	⇒ 80287 内部状態を初期化
FSTSW AX	⇒ AX ← SW
OR AL,AL	⇒ AL=0 であるかどうかを調べる
JZ GOT_287	⇒ AL=0 のとき GOT_287 ヘジャンプする
SMSW AX	⇒ 80287 をもたない場合
OR AX,04H	⇒ EM ← 1
LMSW AX	
JMP CONTINUE	
GOT_287:SMSW AX	⇒ 80287 をもつ場合
OR AX,02H	MP ← 1
LMSW AX	
CONTINUE:	⇒ 処理を続ける

図 9・12 80287 自動検査プログラム

(iAPX286 Hardware Reference Manual (Order No.210760-001),  
6-8, Fig 6-4 より引用)

## 9 数値演算コプロセッサ 80287

に代入する必要がある。80286 の TSS はオフセット 0 からオフセット 43 までが定義されており、特に 80287 のレジスタ状態を保存するための領域は定義されていないが、TSS のリミットを変更し、オフセット 44 からの領域を使用すればよい。

タイプ 7 の割り込みによって、以上のような処理を実行してから、IRET 命令を実行すれば、割り込みの原因となった ESC 命令または WAIT 命令を新しいタスクにおいて再実行することができる。なお、後の処理において再びタスクスイッチを判断するため、割り込み処理の中で CLTS 命令を使用して、TS を 0 に戻しておく必要がある。

〔3〕 **80287 エミュレータ** もし、浮動小数点演算を高速で実行する必要があるれば、80287 を使用せずに、同等の機能をもつ手続きを利用することによって浮動小数点演算を実現してもかまわない。このとき、EM=1、MP=0 にした状態で ESC 命令を実行すれば、タイプ 7 の割り込みを発生するので、割り込み処理によって 80287 のエミュレータを実現することができる。

〔4〕 **MSW の初期設定** 以上のように 80287 を使用するか、しないかによって MSW の MP、EM の 2 ビットを初期設定しなければならない。図 9・11 に 80287 の存在を検査する 1 つの方法と、図 9・12 に MSW の MP、EM を初期設定するプログラムを示す。すなわち、図 9・11 に示したように、80286 と 80287 を接続するデータバスの  $D_7$  を抵抗でプルアップしておき、図 9・12 に示したプログラムで、80287 のレジスタの状態を初期化する命令 FNINIT を実行してから、FSTSW 命令を使用して 80287 のステータスワードの値を AX に代入する。初期化後の 80287 のステータスレジスタの下位 8 ビットは 0 であるから、もし 80287 が存在すれば、AL は 0 である。これに対して、80287 が存在しなければ、少なくともデータバスの  $D_7$  はプルアップされているので、AL には必ず 0 以外の値が代入される。したがって、AL の値が 0 であるかどうかを判定することによって、80287 の状態に応じた MSW の初期設定を実現することができる。

# 10. プログラム開発

インテル社が提供するシステム開発環境を知ること、80286のプロテクトモードで実行されるシステムを開発するうえで有益である。プロテクトモードでは、GDT, LDT, IDT, TSSなどの各種のテーブルを定義し、またセグメントディスクリプタ、ゲートなどの各種のディスクリプタをテーブルに定義しなければならない。さらに、プログラム中で参照するセレクトが間違いなく、対応するディスクリプタを指定しなければならない。もちろん、ディスクリプタには間違いのない値を初期設定しなければならない。アセンブラ、リンカ程度のツールだけでこれだけの作業を行うためには、超人的な注意力が必要である。80286のプロテクトモードの特性を十分に生かしたシステムを開発することは、すなわちマルチタスクで動作するリアルタイム OS を設計することになるのである。しかし、8086の時代におけるセグメンテーションの問題についても同様だが、このような扱いのうえでの困難さは高機能のマクロアセンブラ、高機能のユーティリティプログラムによって十分解決できる。

### 10-1 開発システム

現在 80286 の開発システムとしては、図 10・1 に示すインテル社の MDS シリーズ IV などがある。プログラム言語には、80286 のアセンブリ言語 ASM 286 の他に、PL/M-286、PASCAL-286、FORTRAN-286 などの高級言語が用意されている。また、MDS シリーズ IV は RS-232 C ラインで ROM ライタを接続でき、プログラムの ROM 化までの開発をスムーズに行うためのソフトウェアが用意されている。シリーズ IV 自体は 8086 で動作するシステムであるが、80286 のシミュレータ SIM 286 のようなソフトウェアも用意されており、簡単なソフトウェアデバッグであればシリーズ IV で実行することができる。

シリーズ IV 以外では、パーソナルコンピュータで **UDI** と呼ばれる一種のソフトウェアインタフェースを介して、ASM 286 などのインテルのソフトウェアを使用することも可能である。

また、ハードウェア、ソフトウェアをゼロの状態から開発するのではなく、図 10・2 に示すシステム 310 のような箱入りの 80286 コンピュータを利用することもできる。システム 310 では、XENIX 286 またはリアルタイム OS である iRMX 286 が用意されており、ユーザは応用プログラムを書くだけでよい。



図 10・1 MDS シリーズ IV



図 10・2 システム 310

## 10-2 システム開発とユーティリティプログラム

〔1〕 **プログラム開発環境** ASM 286 などのトランスレータを使用して、ソースプログラムを再配置形式オブジェクトプログラムに変換した後、**BND 286** (バインダと呼ぶ)、**BLD 286** (ビルダと呼ぶ) などのユーティリティプログラムを使用して、実行可能なプログラムを作成する。ここで、バインダは別々のファイルに作成したオブジェクトプログラムを1つに結合するプログラムで、リンカに相当するユーティリティプログラムである。また、ビルダはセグメントの絶対アドレスを定義するいわゆるロケータに相当するユーティリティプログラムである。ただし、ビルダの働きはロケータの機能だけではない。プログラマはシステムで使用するセグメントディスクリプタ、各種のゲート、TSS, GDT, LDT, IDT などの定義を **ビルダプログラム** と呼ばれる書式で、PL/M などのプログラムのソースファイルを作成するのと同様に、ファイルに作っておけば、後はビルダ BLD 286 が

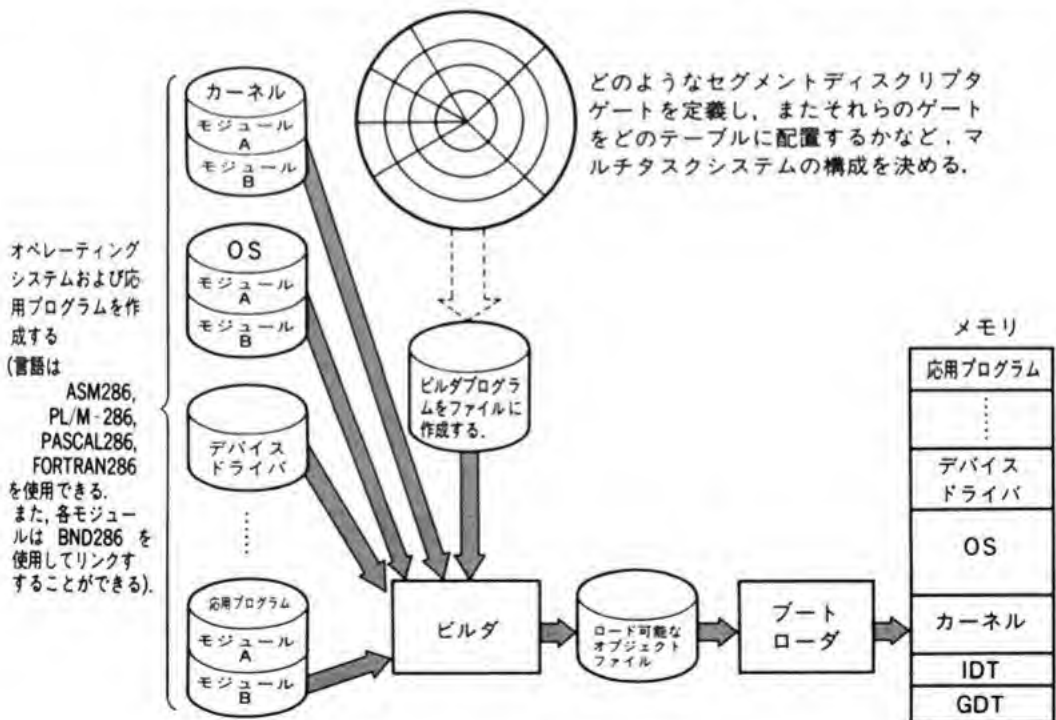


図 10-3 静的システムの開発

(iPX286 Operating Systems Writer's Guide (Order No. 121960-001),  
1-10, Fig 1-8 より引用)

## 10 プログラム開発

ビルダプログラムをコンパイルしてから、ディスクリプタなどのデータをユーザプログラムにリンクする。このとき、ビルダはビルダプログラムの中に明らかな定義の誤りを発見すればエラーメッセージを出力し、またユーザプログラムの定義とビルダプログラムの定義の間に不合理と思われるような点を発見したときは注意メッセージを出力する。このようにディスクリプタ、ゲートなどの 80286 のプロテクトモードシステムに必要なデータをビルダプログラムでシンボリックに定義できるため、かなりのエラーを BLD 286 を実行する段階で排除することができる。

〔2〕 **システム開発の流れ** 次に、典型的な 2 つの例についてシステム開発の流れを考える。

（a） **静的システムの開発** 図 10・3 に静的システムの開発の流れを示す。ここでいう**静的システム**とは図に示すように、システムの電源を投入した後、ブートローダと呼ばれるプログラムがカーネル、デバイスドライバなどの OS をリンクした応用プログラムをディスクファイルからメモリに転送して実行するようなシステムである。ブートローダは ROM メモリに記録されていると考えればよい。このようなシステムでは実行の間 OS も応用プログラムもメモリに常駐する。FA（ファクトリオートメーション）などにおける装置の制御システムにこのような形態が多いだろう。

ブートローダによって読まれるオブジェクトファイルにはセグメント、ディスクリプタテーブルなどの配置アドレスが定義されている。このような絶対番地形式のファイルを作成するユーティリティプログラムがビルダである。ビルダは OS のカーネル、デバイスドライバなどの手続きと応用プログラム、さらにビルダプログラムで定義した GDT, IDT, TSS などをリンクしてから絶対番地形式のオブジェクトファイルを作る。

（b） **動的システムの開発** 動的システムの開発の流れを図 10・4 に示す。**動的システム**とは最初 OS がブートローダによってメモリに転送された後、メモリに常駐する OS の管理の下でさまざまな応用プログラムが実行されるようなシステムである。このシステムでは OS がメモリの空き領域を管理し、応用プログラム自体は再配置形式で作成しておき、プログラムを配置するメモリのアドレスを OS のローダが決める。このような 1 つの応用プログラムを定義した再配置形式のオブジェクトファイルを作るためにバインダ BND 286 を使用する。バインダはい

## 10-2 システム開発とユーティリティプログラム

くつかの再配置形式のオブジェクトファイルをリンクして、実行可能なオブジェクトファイルを作成する。このときビルダが作った**エクスポートファイル**もリンクすることができる。ビルダは作成するプログラムの中で使用されているゲートのセクタ、変数のセクタとオフセットなどの情報だけをエクスポートファイルと呼ばれるファイルを出力することができる。ユーザは、このエクスポートファイルをリンクすることによって、OS 内で定義された手続きを引用するようなプログラムを作成することができる。

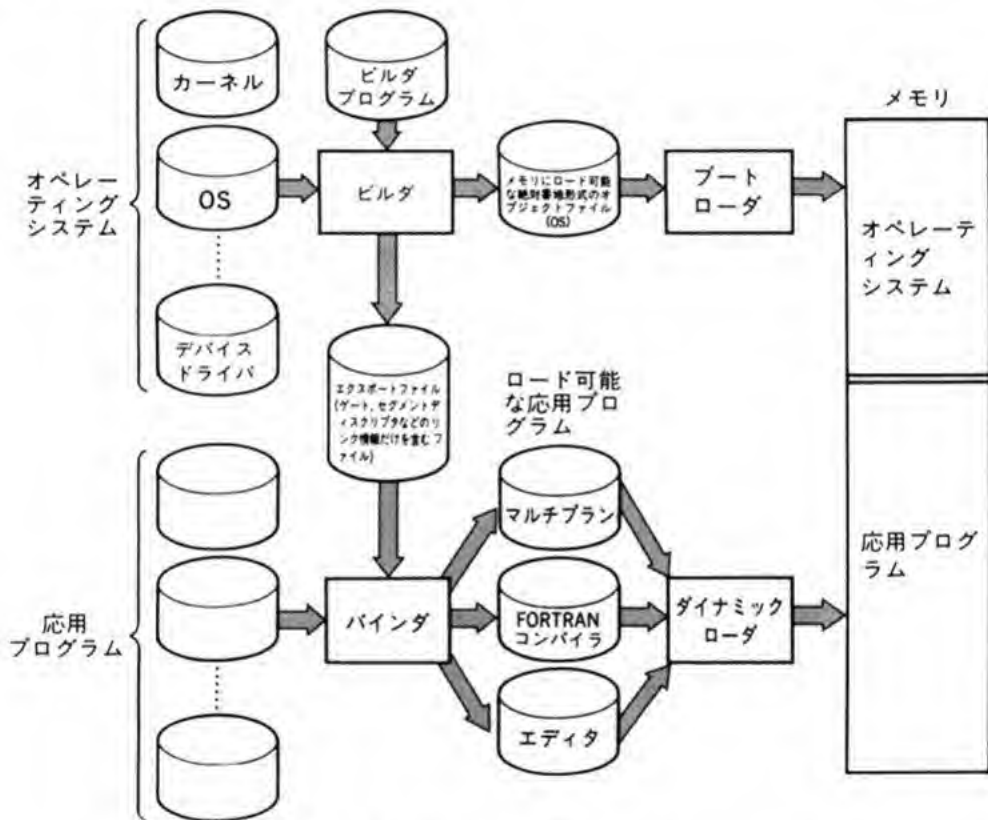


図 10・4 動的システムの開発

(iPX286 Operating Systems Writer's Guide (Order No.121960-001),  
1-11, Fig 1-9 より引用)

## 10-3 デバッグ

〔1〕 デバッグツール 一般にマルチタスクシステムのデバッグは困難である。マルチタスクシステムに限らずプログラムデバッグには王道というものはなく、結局はモジュール化による階層的なシステム開発が最良の方法であるように思う。しかし、そのうえで使いやすいデバッグツールを使用することは有効である。デバッグ用の道具には SIM 286 のようなソフトウェアの論理エラーをデバッグするソフトウェアデバッガもあるが、マイクロコンピュータのハードウェアおよびソフトウェアの開発には ICE（一般にアイスと読まれる）と呼ばれるデバッグツールが利用される。

図 10・5 にインテル社が提供する I<sup>2</sup>ICE（アイスクェアアイスと読まれる）を示す。I<sup>2</sup>ICE は、パーソナリティモジュールと呼ばれる部分を交換することによって 8086, 80186, 80286 のデバッグをすることができる。ICE はインサーキットエミュレータを意味し、名前のようにハードウェアシステムのエミュレーションを行う機能をもつ。



図 10・5 I<sup>2</sup>ICE

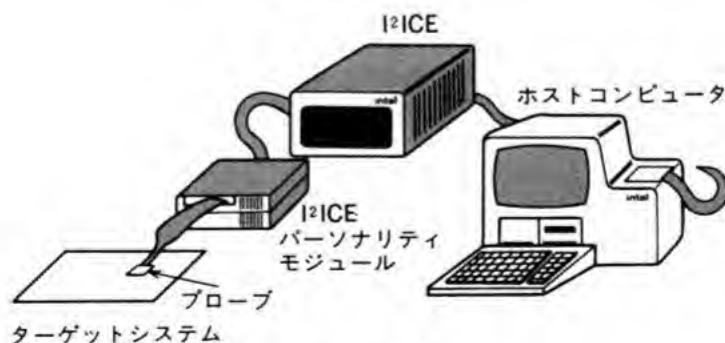


図 10・6 I<sup>2</sup>ICE の使用

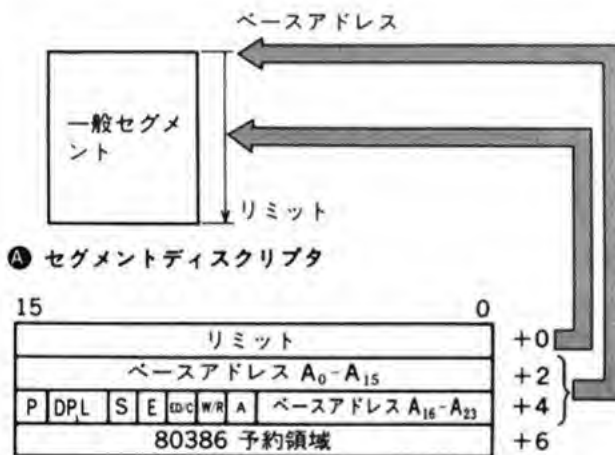
たとえば、I<sup>2</sup>ICE の場合は図 10・6 に示すように、I<sup>2</sup>ICE のプローブをターゲットシステム（最終的にソフトウェアを実行するハードウェアシステム）の 80286 のソケットに差し込み、ホストコンピュータからターゲットシステムのメモリにプログラムを配置して実行することができる。また、ターゲットシステムのハードウェアが未完成の状態であっても、I<sup>2</sup>ICE のもつメモリまたはホストコンピュータのもつメモリを代用して、プログラムのデバッグを行うことができる。

〔2〕 **プロテクトモードのデバッグ** 80286 はセグメントキャッシュの値による厳密なメモリ管理を実行する。さらにプロテクトモードにおいてはセグメントキャッシュと GDT, LDT に定義されるセグメントディスクリプタの関係が 1 対 1 に対応し、80286 はディスクリプタテーブルの値によるメモリ管理または TSS によるタスク管理を実行する。しかし、プログラムのデバッグを実行するときには、これらの保護のための壁をすべて透明にして、80286 のすべての状態をモニタしたい場合がある。そのために、I<sup>2</sup>ICE がもつ 80286 は一般の 80286 とは異なり、保護機能ははずせるようになっている。したがって、PCHECK という I<sup>2</sup>ICE の内部スイッチ変数を OFF にすれば、I<sup>2</sup>ICE の 80286 はデバッグの対象となるプログラムには保護を実行するが、ホストコンピュータの前でデバッグ作業をするオペレータは、本来は見るできないキャッシュの中の状態をも CRT に表示してモニタすることができる。さらに、キャッシュの中のベースアドレス、リミット、アクセスライトの値を直接に書き換えることさえできるのである。



# I. セグメントおよびディスクリプタのまとめ

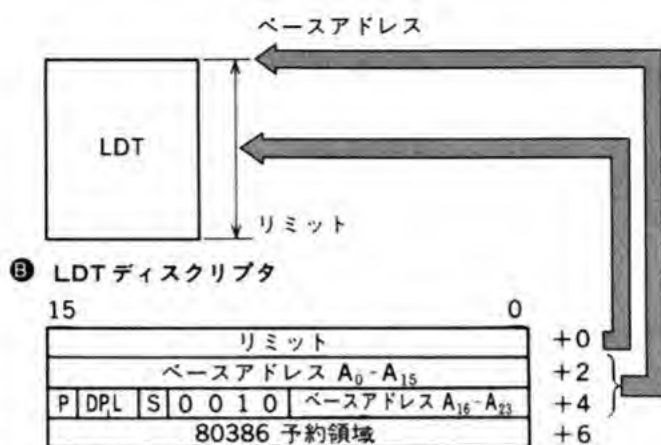
図は、80286 システムで定義されるセグメントおよびディスクリプタの種類をまとめたものである。



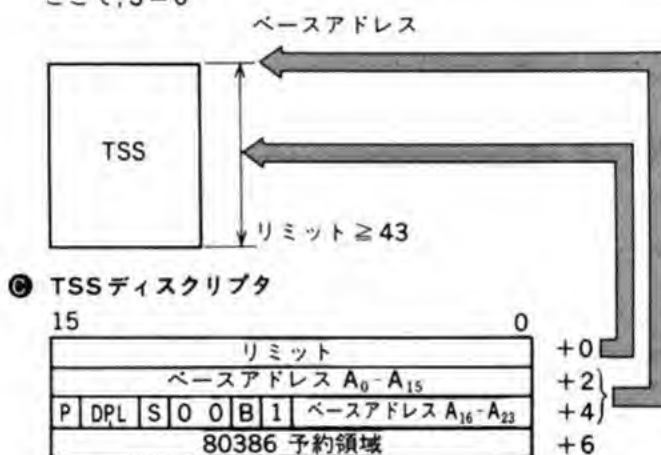
ここで、 $S=1$

セグメントディスクリプタの種類はE, ED/C, W/Rの3ビットによって決まる

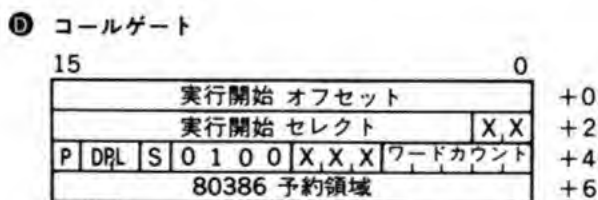
<p>E = 0 のとき このとき</p> <p>ED = 0</p> <p>ED = 1</p> <p>W = 0</p> <p>W = 1</p>	<p>実行不可，リード可</p> <p><math>0 \leq \text{オフセット} \leq \text{リミットの領域だけ使用可}</math></p> <p><math>\text{リミット} + 1 \leq \text{オフセット} \leq 0\text{FFFFH}</math> の領域だけ使用可</p> <p>ライト不可</p> <p>ライト可</p>
<p>E = 1 のとき このとき</p> <p>C = 0</p> <p>C = 1</p> <p>R = 0</p> <p>R = 1</p>	<p>実行可，ライト不可</p> <p>ノンコンフォーミングコードセグメント (一般特権規則を適用)</p> <p>コンフォーミングコードセグメント (例外特権規則を適用)</p> <p>リード不可</p> <p>リード可</p>



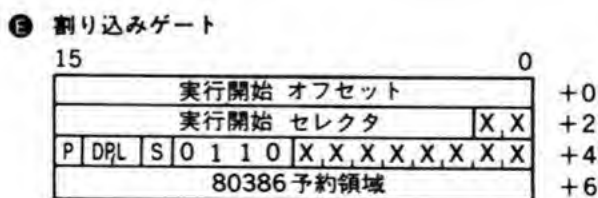
ここで,  $S=0$



ここで,  $S=0$



ここで,  $S=0$



ここで,  $S=0$

## ⑥ タスクゲート

15																0	
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	+0	
TSS セレクタ																	+2
P	DPL	S	0	1	0	1	X	X	X	X	X	X	X	X	X	+4	
80386 予約領域																	+6

ここで, S = 0

## ⑦ トラップゲート

15													0	
実行開始 オフセット													+0	
実行開始 セレクタ												X,X	+2	
P	DPL	S	0	1	1	1	X	X	X	X	X	X	X	+4
80386 予約領域													+6	

ここで, S = 0

各ディスクリプタを定義できるディスクリプタテーブルの関係

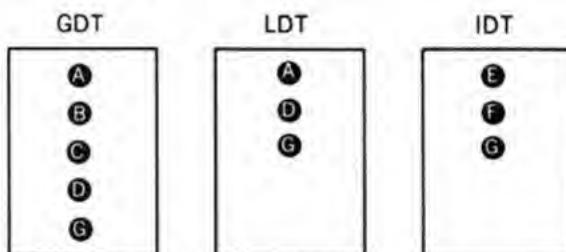


図 ディスクリプタの種類

## II. 80286 命令コード

表付・1 に 80286 の命令コードを示す。また、命令コードの中の mod, disp, r/m, reg の識別を表付・2 から表付・5 に示す。80286 が実行する重要な保護動作、タスクスイッチの動作などの処理の流れをリスト (a) から (h) までに示す。このリストにおいて、DT は GDT または LDT のディスクリプタテーブルを表し、DT (セクタ) によってセクタによって識別されるテーブル内の特定のディスプリクタを表すものとする。さらに、ディスクリプタ内の P, DPL, E, C, W, LIMIT などの各フィールドは DT (セクタ)、P, DT (セクタ)、DPL, DT (セクタ)、LIMIT のように表している。また、DTR は GDTR または LDTR のどちらかのディスクリプタテーブルレジスタを表す。GDTR か LDTR かの区別は、その状況で扱っているセクタの TI ビットによって区別できる。

また、80286 が発生する保護エラーは

タイプ 13 … GP error

タイプ 11 … NP error

タイプ 12 … SS error

タイプ 10 … TS error

のように表現している。さらに続く括弧の中にスタックに PUSH されるエラーコードを指定する。

表 付・1 80286 命令コード

命令ニーモニックとコード	コ メ ン ト	リアル モード	プロテクトモード
<b>MOV (move)</b> <div>1 000100 w mod reg r/m</div> <div>1 000101 w mod reg r/m</div> <div>1 100011 w mod 000 r/m data data if w=1</div> <div>1 011w reg data data if w=1</div> <div>1 010000 w addr-low addr-high</div> <div>1 010001 w addr-low addr-high</div> <div>1 000111 0 mod 0 reg r/m</div> <div>1 000110 0 mod 0 reg r/m</div>	reg/mem ← reg reg ← reg/mem reg/mem ← imm reg ← imm accum ← mem mem ← accum sreg ← reg/mem reg/mem ← sreg	○	○
<b>PUSH (push)</b> <div>1 111111 1 mod 110 r/m</div> <div>0 1010 reg</div> <div>0 00reg11 0</div> <div>0 11010s 0 data data if s=0</div>	push mem push reg push sreg push imm	○	○
<b>PUSHA (push all)</b> <div>0 110000 0</div>	すべてのワード汎用レジスタを スタックに PUSH する	○	○

命令ニーモニックとコード	コ メ ン ト	リアル モード	プロテクトモード
<b>POP (pop)</b> <div>1 000111 1 mod 000 r/m</div> <div>0 1011 reg</div> <div>0 00reg11 1 (reg ≐ 01)</div>	pop mem pop reg pop sreg	○	○  リスト (a) 参照
<b>POPA (pop all)</b> <div>0 110000 1</div>	スタックからすべてのワード 汎用レジスタをPOP する。	○	○
<b>XCHG (exchange)</b> <div>1 000011 w mod reg r/m</div> <div>1 0010 reg</div>	reg/mem $\leftrightarrow$ reg reg $\leftrightarrow$ accum	○	○
<b>IN (in)</b> <div>1 110010 w port</div> <div>1 110110 w</div>	accum $\leftarrow$ port で指定した I/O accum $\leftarrow$ DX で指定した I/O	○	○
<b>OUT (out)</b> <div>1 110011 w port</div> <div>1 110111 w</div>	port で指定した I/O $\leftarrow$ accum DX で指定した I/O $\leftarrow$ accum	○	○
<b>XLAT (translate byte to AL)</b> <div>1 101011 1</div>	オフセット=(BX+AL)のメモリの 1 バイトの値をAL に代入する。	○	○
<b>LEA (load effective address to register)</b> <div>1 000110 1 mod reg r/m</div>	reg で指定したレジスタに r/m で指定したメモリのオフセット を代入する。	○	○
<b>LDS (load pointer to DS: reg)</b> <div>1 100010 1 mod reg r/m (mod ≐ 11)</div>	r/m で指定したメモリの上位ワ ードの値を DS に代入し、下位 ワードを reg で指定したレジ スタに代入する。	○	○ リスト (a) 参照
<b>LES (load pointer to ES: reg)</b> <div>1 100010 0 mod reg r/m (mod ≐ 11)</div>	r/m で指定したメモリの上位ワ ードの値を ES に代入し、下位 ワードを reg で指定したレジ スタに代入する。	○	○ リスト (a) 参照
<b>LAHF (load AH with flags)</b> <div>1 001111 1</div>	AH $\leftarrow$ フラグレジスタの下位バ イト	○	○
<b>SAHF (store AH into flags)</b> <div>1 001111 0</div>	フラグレジスタの下位バイト $\leftarrow$ AH	○	○
<b>PUSHF (push flags)</b> <div>1 001110 0</div>	フラグレジスタの値をスタック に PUSH する。	○	○
<b>POPF (pop flags)</b> <div>1 001110 1</div>	スタックから POP した値をフ ラグレジスタに代入する。	○	○

命令ニーモニックとコード	コ メ ン ト	リアル モード	プロテクトモード
ADD (add) 0 0000d w mod reg r/m  1 0000s w mod 000 r/m data data if s w=01  0 000010 w data data if w=1	(d=1) $\text{reg} \leftarrow \text{reg} + \text{reg}/\text{mem}$ (d=0) $\text{reg}/\text{mem} \leftarrow \text{reg}/\text{mem} + \text{reg}$ $\text{reg}/\text{mem} \leftarrow \text{reg}/\text{mem} + \text{imm}$ $\text{accum} \leftarrow \text{accum} + \text{imm}$	○	○
ADC (add with carry) 0 00100d w mod reg r/m  1 0000s w mod 010 r/m data data if s w=01  0 001010 w data data if w=1	(d=1) $\text{reg} \leftarrow \text{reg} + \text{reg}/\text{mem} + \text{carry flag}$ (d=0) $\text{reg}/\text{mem} \leftarrow \text{reg}/\text{mem} + \text{reg} + \text{carry flag}$ $\text{reg}/\text{mem} \leftarrow \text{reg}/\text{mem} + \text{imm} + \text{carry flag}$ $\text{accum} \leftarrow \text{accum} + \text{imm} + \text{carry flag}$	○	○
INC (increment) 1 111111 w mod 000 r/m 0 1000 reg	$\text{reg}/\text{mem} \leftarrow \text{reg}/\text{mem} + 1$ $\text{reg} \leftarrow \text{reg} + 1$	○	○
SUB (subtract) 0 01010d w mod reg r/m  1 0000s w mod 101 r/m data data if s w=01  0 010110 w data data if w=1	(d=1) $\text{reg} \leftarrow \text{reg} - \text{reg}/\text{mem}$ (d=0) $\text{reg}/\text{mem} \leftarrow \text{reg}/\text{mem} - \text{reg}$ $\text{reg}/\text{mem} \leftarrow \text{reg}/\text{mem} - \text{imm}$ $\text{accum} \leftarrow \text{accum} - \text{imm}$	○	○
SBB (subtract with borrow) 0 00110d w mod reg r/m  1 0000s w mod 011 r/m data data if s w=01  0 001110 w data data if w=1	(d=1) $\text{reg} \leftarrow \text{reg}/\text{mem} - \text{carry flag}$ (d=0) $\text{reg}/\text{mem} \leftarrow \text{reg}/\text{mem} - \text{reg} - \text{carry flag}$ $\text{reg}/\text{mem} \leftarrow \text{reg}/\text{mem} - \text{imm} - \text{carry flag}$ $\text{accum} \leftarrow \text{accum} - \text{imm} - \text{carry flag}$	○	○
DEC (decrement) 1 111111 w mod 001 r/m 0 1001 reg	$\text{reg}/\text{mem} \leftarrow \text{reg}/\text{mem} - 1$ $\text{reg} \leftarrow \text{reg} - 1$	○	○
CMP (compare) 0 011101 w mod reg r/m 0 011100 w mod reg r/m 1 0000s w mod 111 r/m data data if s w=01 0 011110 w data data if w=1	$\text{reg} - \text{reg}/\text{mem}$ $\text{reg}/\text{mem} - \text{reg}$ $\text{reg}/\text{mem} - \text{imm}$ $\text{accum} - \text{imm}$	○	○
NEG (negate) 1 111011 w mod 011 r/m	2の補数をとる。	○	○
AAA (ASCII adjust for add) 0 011011 1	ALにある純2進数をアンパクトBCDに補正する。純2進加算を実行した後に使用する。	○	○

命令ニーモニックとコード	コ メ ン ト	リアル モード	プロテクトモード
DAA (decimal adjust for add) 0 010011 1	AL にある純 2 進をバケット BCD に補正する。純 2 進加算を実行した後に使用する。	○	○
AAS (ASCII adjust for subtract) 0 011111 1	AL にある純 2 進をアンバケット BCD に補正する。純 2 進減算を実行した後に使用する。	○	○
DAS (decimal adjust for subtract) 0 010111 1	AL にある純 2 進をバケット BCD に補正する。純 2 進減算を実行した後に使用する。	○	○
MUL (unsigned multiply) 1 111011 w   mod 100 r/m	(w=0) AX←AL*reg/mem (w=1) DX, AX←AX *reg/mem	○	○
IMUL (integer multiply) 1 111011 w   mod 101 r/m	(w=0) AX←AL*reg/mem (w=1) DX, AX←AX *reg/mem	○	○
IMUL (integer immediate multiply) 0 11010s 1   mod reg r/m   data   data if s=0	reg←reg/mem*imm	○	○
DIV (unsigned divide) 1 111011 w   mod 110 r/m	(w=0) $\begin{cases} \text{AL} \leftarrow \text{AX} \\ \div \text{reg/mem} \\ \text{AH} \leftarrow \text{AX MOD} \\ \text{reg/mem} \end{cases}$ (w=1) $\begin{cases} \text{AX} \leftarrow \text{DX, AX} \\ \div \text{reg/mem} \\ \text{DX} \leftarrow \text{DX, AX MOD} \\ \text{reg/mem} \end{cases}$	○	○
IDIV (integer divide) 1 111011 w   mod 111 r/m	(w=0) $\begin{cases} \text{AL} \leftarrow \text{AX} \\ \div \text{reg/mem} \\ \text{AH} \leftarrow \text{AX MOD} \\ \text{reg/mem} \end{cases}$ (w=1) $\begin{cases} \text{AX} \leftarrow \text{DX, AX} \\ \div \text{reg/mem} \\ \text{DX} \leftarrow \text{DX, AX MOD} \\ \text{reg/mem} \end{cases}$	○	○
AAM (ASCII adjust for multiply) 1 101010 0   0 000101 0	アンバケット BCD の乗算のための補正命令。乗算命令を実行してから使用する。	○	○
AAD (ASCII adjust for divide) 1 101010 1   0 000101 0	アンバケット BCD の除算のための補正命令。除算命令を実行する前に使用する。	○	○
CBW (convert byte to word) 1 001100 0	AL にある符号付き整数を AX に拡張する。	○	○

命令ニーモニックとコード	コ メ ン ト	リアル モード	プロテクトモード
CWD (convert word to double word) 1 001100 1	AXにある符号付き整数をDX, AXに拡張する.	○	○
shift/rotate 1 101000 w mod TTT r/m 1 101001 w mod TTT r/m 1 100000 w mod TTT r/m count  TTT 命令ニーモニック 0 0 0 ROL 0 0 1 ROR 0 1 0 RCL 0 1 1 RCR 1 0 0 SHL/SAL 1 0 1 SHR 1 1 1 SAR	reg/memを1ビットだけ シフト/回転する. reg/memをCLビットだけ シフト/回転する. reg/memをimmビットだけ シフト/回転する.	○	○
AND (and) 0 01000d w mod reg r/m  1 000000 w mod 100 r/m data data if w=1 0 010010 w data data if w=1	(d=1) reg←reg AND reg/mem (d=0) reg/mem←reg/mem AND reg reg/mem←reg/mem AND imm accum←accum AND imm	○	○
TEST (test) 1 000010 w mod reg r/m 1 111011 w mod 000 r/m data data if w=1 1 010100 w data data if w=1	reg/mem AND reg reg/mem AND imm accum AND imm	○	○
OR (or) 0 00010d w mod reg r/m  1 000000 w mod 001 r/m data data if w=1 0 000110 w data data if w=1	(d=1) reg←reg OR reg/mem (d=0) reg/mem←reg/mem OR reg reg←reg OR imm accum←accum OR imm	○	○
XOR (exclusive or) 0 01100d w mod reg r/m  1 000000 w mod 110 r/m data data if w=1 0 011010 w data data if w=1	(d=1) reg←reg XOR reg/mem (d=0) reg/mem←reg/mem XOR reg reg/mem←reg/mem XOR imm accum←accum XOR imm	○	○
NOT (not) 1 111011 w mod 010 r/m	NOT reg/mem	○	○

命令ニーモニックとコード	コ メ ン ト	リアル モード	プロテクトモード
MOVS (move string) 1 010010 w CMPS (compare string) 1 010011 w SCAS (scan string) 1 010111 w LODS (load string) 1 010110 w STOS (store string) 1 010101 w INS (in string) 0 110110 w OUTS (out string) 0 110111 w	スtring命令	○	○
REP (repeat prefix) 1 111001 0 REPE/REPZ 1 111001 1 REPNE/REPNZ 1 111001 0	REPの直後にあるString命令をCX回だけ繰り返す。 REPEまたはREPZの直後にあるString命令をCX回だけ、またはZF=0になるまで繰り返す。 REPNEまたはREPNZの直後にあるString命令をCX回だけ、またはZF=1になるまで繰り返す。	○	○
CALL (call) 1 110100 0    disp-low    disp-high 1 111111 1    mod 010 r/m 1 001101 0    segment offset segment selector 1 111111 1    mod 011 r/m    (mod ≠ 11)	直接 near call reg/mem 間接 near call 直接 far call 間接 far call	○	○ リスト (d) 参照 リスト (d) 参照
RET (return) 1 100001 1 1 100001 0    data-low    data-high 1 100101 1 1 100101 0    data-low    data-high	near return near return (SP ← SP + imm を伴う) far return far return (SP ← SP + imm を伴う)	○	○ リスト (e) 参照
JMP (jump) 1 110101 1    disp-low 1 110100 1    disp-low    disp-high 1 111111 1    mod 100 r/m 1 110101 0    segment offset segment selector 1 111111 1    mod 101 r/m    (mod ≠ 11)	short jump 直接 near jump reg/mem 間接 near jump 直接 far jump 間接 far jump	○	○ リスト (c) 参照 リスト (c) 参照

命令ニーモニックとコード	コ メ ン ト	リアル モード	プロテクトモード
<p>JE/JZ (jump on equal/zero)</p> <div data-bbox="162 272 465 305">0 111010 0      disp</div> <p>JL/JNGE (jump on less/not greater or equal)</p> <div data-bbox="162 343 465 376">0 111110 0      disp</div> <p>JLE/JNG (jump on less or equal/not greater)</p> <div data-bbox="162 415 465 448">0 111111 0      disp</div> <p>JB/JNAE (jump on below/not above or equal)</p> <div data-bbox="162 486 465 519">0 111001 0      disp</div> <p>JBE/JNA (jump on below or equal/not above)</p> <div data-bbox="162 558 465 591">0 111011 0      disp</div> <p>JP/JPE (jump on parity/parity even)</p> <div data-bbox="162 629 465 662">0 111101 0      disp</div> <p>JO (jump on overflow)</p> <div data-bbox="162 701 465 734">0 111000 0      disp</div> <p>JS (jump on sign)</p> <div data-bbox="162 772 465 805">0 111100 0      disp</div> <p>JNE/JNZ (jump on not equal/not zero)</p> <div data-bbox="162 844 465 877">0 111010 1      disp</div> <p>JNL/JGE (jump on not less/greater or equal)</p> <div data-bbox="162 915 465 948">0 111110 1      disp</div> <p>JNLE/JG (jump on not less or equal/greater)</p> <div data-bbox="162 987 465 1020">0 111111 1      disp</div> <p>JNB/JAE (jump on not below/above or equal)</p> <div data-bbox="162 1058 465 1091">0 111001 1      disp</div> <p>JNBE/JA (jump on not below or equal/above)</p> <div data-bbox="162 1130 465 1163">0 111011 1      disp</div> <p>JNP/JPO (jump on not parity/parity odd)</p> <div data-bbox="162 1201 465 1234">0 111101 1      disp</div> <p>JNO (jump on not overflow)</p> <div data-bbox="162 1273 465 1306">0 111000 1      disp</div> <p>JNS (jump on not sign)</p> <div data-bbox="162 1344 465 1377">0 111100 1      disp</div> <p>LOOP (loop)</p> <div data-bbox="162 1416 465 1449">1 110001 0      disp</div> <p>LOOPZ/LOOPE (loop while zero/equal)</p> <div data-bbox="162 1487 465 1520">1 110000 1      disp</div> <p>LOOPNZ/LOOPNE (loop while not zero/equal)</p> <div data-bbox="162 1559 465 1591">1 110000 0      disp</div> <p>JCXZ (jump on CX zero)</p> <div data-bbox="162 1630 465 1663">1 110001 1      disp</div>	<p>ZF=1 のときジャンプする。</p> <p>(SF XOR OF)=1 のときジャンプする。</p> <p>((SF XOR OF) OR ZF)=1 のときジャンプする。</p> <p>CF=1 のときジャンプする。</p> <p>(CF OR ZF)=1 のときジャンプする。</p> <p>PF=1 のときジャンプする。</p> <p>OF=1 のときジャンプする。</p> <p>SF=1 のときジャンプする。</p> <p>ZF=0 のときジャンプする。</p> <p>(SF XOR OF)=0 のときジャンプする。</p> <p>((SF XOR OF) OR ZF)=0 のときジャンプする。</p> <p>CF=0 のときジャンプする。</p> <p>(CF OR ZF)=0 のときジャンプする。</p> <p>PF=0 のときジャンプする。</p> <p>OF=0 のときジャンプする。</p> <p>SF=1 のときジャンプする。</p> <p>CX←CX-1 CX=0 でなければジャンプする。 (IP←IP+disp)</p> <p>CX←CX-1 CX≠0 かつ ZF=1 のときジャンプする。</p> <p>CX←CX-1 CX≠0 かつ ZF=0 のときジャンプする。</p> <p>CX=0 のときジャンプする。</p>	○	○
<p>ENTER (enter)</p> <div data-bbox="162 1657 770 1690">1 100100 0      data-low      data-high      L</div>		○	○
<p>LEAVE (leave)</p> <div data-bbox="162 1734 317 1767">1 100100 1</div>		○	○

命令ニーモニックとコード	コ メ ン ト	リアル モード	プロテクトモード
INT (interrupt) <div>1 100110 1      type</div> <div>1 100110 0</div>	INT type INT 3	○	○ (リスト (f) 参照)
INTO (interrupt on overflow) <div>1 100111 0</div>	OF=1 のときタイプ4の割り 込みに入る。	○	○ (リスト (f) 参照)
IRET (interrupt return) <div>1 100111 1</div>	(NT=0) 割り込みリターン命令 (NT=1) タスクスイッチ	○ ×	○ ○ (リスト (g) 参照) (リスト (g) 参照)
BOUND (bound) <div>0 110001 0      mod reg r/m</div>		○	○
CLC (clear carry) <div>1 111100 0</div>	CF ← 0	○	○
CMC (complement carry) <div>1 111010 1</div>	(CF=0) CF ← 1 (CF=1) CF ← 0	○	○
STC (set carry) <div>1 111100 1</div>	CF ← 1	○	○
CLD (clear direction) <div>1 111110 0</div>	DF ← 0	○	○
STD (set direction) <div>1 111110 1</div>	DF ← 1	○	○
CLI (clear interrupt) <div>1 111101 0</div>	IF ← 0	○	○ if IOPL ≥ CPL else GPError (0)
STI (set interrupt) <div>1 111101 1</div>	IF ← 1	○	○ if IOPL ≥ CPL else GPError (0)
HLT (halt) <div>1 111010 0</div>	ホルト命令 80286 の動作を停止する。	○	○ if CPL = 0 else GPError (0)
WAIT (wait) <div>1 001101 1</div>	BUSY=Low のとき、再び BUSY=High になるまで次の 命令に進まない。	○	○
LOCK (lock prefix) <div>1 111000 0</div>	LOCK の直後の命令実行におけ るバスサイクルにおいて LOCK =Low となる。	○	○
CLTS (clear task switch flag) <div>0 000111 1      0 000011 0</div>	MSW の TS ← 0	○	○ if CPL = 0 else GPError (0)
ESC (80287 escape) <div>1 1011TT T      mod LLL r/m</div>	80287 の命令 TTT LLL のコードによって 80 287 の各種の命令を表す。	○	○

命令ニーモニックとコード	コ メ ン ト	リアル モード	プロテクトモード
SEG (segment override prefix) 0 01reg11 0	SEG の直後の命令において、メモリ参照に使用するセグメントレジスタを指定する。	○	○
LGDTR (load GDTR) 0 000111 1 0 000000 1 mod 010 r/m	GDTR ← mem	○	○ if CPL=0 else GPError (0)
SGDTR (store GDTR) 0 000111 1 0 000000 1 mod 000 r/m	mem ← GDTR	○	○
LIDTR (load IDTR) 0 000111 1 0 000000 1 mod 011 r/m	IDTR ← mem	○	○ if CPL=0 else GPError (0)
SIDT (store IDTR) 0 000111 1 0 000000 1 mod 001 r/m	mem ← IDTR	○	○
LLDT (load LDTR) 0 000111 1 0 000000 0 mod 010 r/m	LDTR ← reg/mem LDTR.CACHE ← GDT (LDTR)	×	○ if CPL=0 else GPError (0)
SLDT (store LDTR) 0 000111 1 0 000000 0 mod 000 r/m	reg/mem ← LDTR	×	○
LTR (load TR) 0 000111 1 0 000000 0 mod 011 r/m	TR ← reg/mem TR.CACHE ← GDT (TR)	×	○ if CPL=0 else GPError (0)
STR (store TR) 0 000111 1 0 000000 0 mod 001 r/m	reg/mem ← TR	×	○
LMSW (load MSW) 0 000111 1 0 000000 1 mod 110 r/m	MSW ← reg/mem	○	○ if CPL=0 else GPError (0)
SMSW (store MSW) 0 000111 1 0 000000 1 mod 100 r/m	reg/mem ← MSW	○	○
LAR (load access right) 0 000111 1 0 000001 0 mod reg r/m	reg/mem で指定されるセクタのディスクリプタのアクセスライトを reg の上位バイトに代入する。	×	○
LSL (load segment limit) 0 000111 1 0 000001 1 mod reg r/m	reg/mem で指定されるセクタのディスクリプタのリミットを reg に代入する。	×	○
ARPL (adjust requested privilege level) 0 000111 1 0 000001 1 mod reg r/m	reg/mem で指定されるセクタのディスクリプタのリミットを reg に代入する。	×	○
ARPL (adjust requested privilege level) 0 110001 1 mod reg r/m		×	○
VERR (verify read access) 0 000111 1 0 000000 0 mod 100 r/m	reg/mem で指定されるセクタのセグメントがリード可能かどうかを検査する。	×	○

VERW (verify write access)			×	○
0 000111 1	0 000000 0	mod 101 r/m		
			reg/mem で指定されるセクタのセグメントがライト可能かどうかを検査する。	

表 付・2 mod のコードと意味

mod	意 味
11	r/m はレジスタを表す。
00	disp を表すコードは 0 バイト
01	1 バイトの符号付きデータで disp を表す。
10	2 バイトの符号付きデータで disp を表す。

ただし、mod=00かつr/m=110 のとき、2 バイトの符号付きデータで disp を表す。

表 付・3 r/m のコードと意味 (mod=00, 01, 10 のとき r/m はメモリオペランドを表す)

r/m	EA (effective address)
000	[BX] + [SI] + disp
001	[BX] + [DI] + disp
010	[BP] + [SI] + disp
011	[BP] + [DI] + disp
100	[SI] + disp
101	[DI] + disp
110	[BP] + disp (ただし mod=00, r/m=110 のときのみ、EA=2 バイトの disp (直接指定) となる)
111	[BX] + disp

メモリ参照に実効的に使用されるオフセットを EA という。

表 付・4 r/m がレジスタを表すとき、また reg の識別

reg, r/m(mod=11)	レジスタ名	
	W = 1	W = 0
000	AX	AL
001	CX	CL
010	DX	DL
011	BX	BL
100	SP	AH
101	BP	CH
110	SI	DH
111	DI	BH

表 付・5 reg がセグメントレジスタを表現するときのコード

sreg	セグメントレジスタ
00	ES
01	CS
10	SS
11	DS

リスト(a) DS, ES にセクタ値 SEL を代入するとき, 80286 は次の処理を実行する.

```

if SEL ≠ ヌルセクタ
  then | if (SEL.INDEX+1)*8-1 ≤ DTR.LIMIT else GPerror(SEL);
        if (DT(SEL).S=0 AND (DT(SEL).E=0 OR DT(SEL).R=1))
          else GPerror(SEL);
        if DT(SEL).E=0 OR (DT(SEL).E=1 AND DT(SEL).C=0)
          then | if DT(SEL).DPL ≥ MAX(CS.CPL, SEL.RPL)
                  else GPerror(SEL);
                 if DT(SEL).P=1 else NPerror(SEL); |
        SREG ← SEL;  (* ここでSREGはDSまたはES *)
        SREG.CACHE ← DT(SEL); |
  else | (* セクタがヌルセクタの場合 *)
        SREG ← 0;  (* DS または ES に 0 を代入する *)
        SREG.CACHE ← invalid; (* キャッシュレジスタを 80286 が使用できないよ
                                うに不正であるという印を付ける *)

```

リスト(b) SS にセクタ値 SEL を代入するとき, 80286 は次の処理を実行する.

```

if SEL ≠ ヌルセクタ else GPerror(0);
if (SEL.INDEX+1)*8-1 ≤ DTR.LIMIT else GPerror(SEL);
if (SEL.RPL=CS.CPL) else GPerror(SEL)
if DT(SEL).S=1 AND DT(SEL).E=0 AND DT(SEL).W=1
  else GPerror(SEL);
if DT(SEL).DPL=CS.CPL else GPerror(SEL);
if DT(SEL).P=1 else SSerror(SEL);
SS ← SEL;
SS.CACHE ← DT(SEL);

```

リスト(c) far JMP 命令を使用して CS の値を NEW\_SEL に, IP を NEW\_IP に変更するとき, 80286 は次の処理を実行する.

```

if 間接 JMP
  then | if SREG.CACHE.E=0 OR (SREG.CACHE.E=1 AND SREG.
                                CACHE.R=1)
          (* ここで, SREG は JMP 命令がメモリを参照するときに使用するセグメントレ
              ジスタを表す *)
          else GPerror(0); |
if NEW_SEL ≠ ヌルセクタ else GPerror(0);
if (NEW_SEL.INDEX+1)*8-1 ≤ DTR.LIMIT
  else GPerror(NEW_SEL);
if DT(NEW_SEL).E=1 AND DT(NEW_SEL).C=1)

```

```

then | (* NEW_SEL の指定するセグメントが特権例外のコードセグメントであるとき *)
    if DT(NEW_SEL).DPL ≤ CS.CPL else GError(NEW_SEL);
    if DT(NEW_SEL).P=1 else NError(NEW_SEL);
    if NEW_IP ≤ DT(NEW_SEL).LIMIT else GError(0);
    CS ← NEW_SEL;
    IP ← NEW_IP;
    CS.CACHE ← DT(NEW_SEL);|
if DT(NEW_SEL).E=1 AND DT(NEW_SEL).C=0
then | (* NEW_SEL の指定するセグメントがノンコンフォーミングコードセグメントである
    場合 *)
    if NEW_SEL.RPL ≤ CS.CPL else GError(NEW_SEL);
    if DT(NEW_SEL).DPL=CS.CPL else GError(NEW_SEL);
    if DT(NEW_SEL).P=1 else NError(NEW_SEL);
    if NEW_IP ≤ DT(NEW_SEL).LIMIT else GError(0);
    CS ← NEW_SEL;
    IP ← NEW_IP;
    CS.CACHE ← DT(NEW_SEL);
    CS.RPL ← SS.CPL;|
if DT(NEW_SEL)=CALL_GATE
then | (* NEW_SEL が指定するディスクリプタがコールゲートである場合 *)
    if CALL_GATE.DPL ≥ CS.CPL else GError(NEW_SEL);
    if CALL_GATE.DPL ≥ NEW_SEL.RPL
        else GError(NEW_SEL);
    if CALL_GATE.P = 1 else NError(NEW_SEL);
    if CALL_GATE.SEL ≠ ヌルセクタ else GError(0);
    if (CALL_GATE.SEL.INDEX+1)*8-1 ≤ DTR.LIMIT
        else GError(CALL_GATE.SEL);
    if DT(CALL_GATE.SEL).E=1
        else GError(CALL_GATE.SEL);
    if DT(CALL_GATE.SEL).C=0
        then | if DT(CALL_GATE.SEL).DPL = CS.CPL
            else GError(CALL_GATE.SEL);|
    if DT(CALL_GATE.SEL).C = 1
        then | if DT(CALL_GATE.SEL).DPL ≤ CS.CPL
            else GError(CALL_GATE.SEL);|
    if DT(CALL_GATE.SEL).P=1
        else NError(CALL_GATE.SEL);
    if CALL_GATE.IP ≤ DT(CALL_GATE.SEL).LIMIT
        else GError(0);
    CS ← CALL_GATE.SEL;
    IP ← CALL_GATE.IP;
    CS.CACHE ← DT(CS);
    CS.RPL ← SS.CPL;|
if DT(NEW_SEL) = TASK_GATE
then | (* NEW_SEL が指定するディスクリプタがタスクゲートである場合 *)

```

```

if TASK_GATE.DPL ≥ CS.CPL else GError(NEW_SEL);
if TASK_GATE.DPL ≥ NEW_SEL.RPL
    else GError(NEW_SEL);
if TASK_GATE.P = 1 else NError(NEW_SEL);
if TASK_GATE.SEL.TI = 0
    else GError(TASK_GATE.SEL);
if (TASK_GATE.SEL.INDEX+1)*8-1 ≤ DTR.LIMIT
    else GError(TASK_GATE.SEL);
if DT(TASK_GATE.SEL).B = 0
    else GError(TASK_GATE.SEL);
if DT(TASK_GATE.SEL).P = 1
    else NError(TASK_GATE.SEL);
TASK_SWITCH;(* TSS セレクタが TASK_GATE.SEL で指定されるタスクに
              スイッチする *)
if IP ≤ CS.CACHE.LIMIT else GError(0);|
if DT(NEW_SEL) = TSS_DESCRIPTOR
    then|(* NEW_SEL が指定するディスクリプタが TSS ディスクリプタである場合 *)
        if DT(NEW_SEL).DPL ≥ CS.CPL else GError(NEW_SEL);
        if DT(NEW_SEL).DPL ≥ NEW_SEL.RPL
            else GError(NEW_SEL);
        if DT(NEW_SEL).B = 0 else GError(NEW_SEL);
        if DT(NEW_SEL).P = 1 else NError(NEW_SEL);
        TASK_SWITCH;(* TSS セレクタが NEW_SEL で指定されるタスクにスイッチす
                      る *)
        if IP ≤ CS.CACHE.LIMIT else GError(0);|

```

リスト(d) far CALL 命令を使用して CS の値を NEW\_SEL に、また IP の値を NEW\_IP に変更するとき、80286 は次の処理を実行する。

```

if 間接 CALL
    then|if SREG.CACHE.E = 0 OR (SREG.CACHE.E=1
        AND SREG.CACHE.R=1)
        (* ここで SREG は CALL 命令がメモリを参照するときに使用するセグメントレジ
          スタを表す *)
        else GError(0);|
if NEW_SEL ≠ ヌルセレクタ else GError(0);
if (NEW_SEL.INDEX+1)*8-1 ≤ DTR.LIMIT else GError
    else GError(NEW_SEL);
if DT(NEW_SEL).E=1 AND DT(NEW_SEL).C=1
    then|(* NEW_SEL の指定するセグメントが特権例外のコードセグメントである場合 *)
        if DT(NEW_SEL).DPL ≤ CS.CPL else GError(NEW_SEL);
        if DT(NEW_SEL).P = 1 else NError(NEW_SEL);
        if (SP-4) ≥ SS.CACHE.LIMIT (* スタックに戻りアドレスを PUSH す
          る空間があるかどうか *)

```

```

    else SSerror(0);
    if NEW_IP ≤ DT(NEW_SEL).LIMIT else GPerror(0);
    CS ← DT(NEW_SEL);
    CS ← NEW_SEL;
    IP ← NEW_IP;|
if DT(NEW_SEL).E=1 AND DT(NEW_SEL).C=0
then | (* NEW_SEL の指定するセグメントがノンコンフォーミングコードセグメントである
    場合 *)
    if NEW_SEL.RPL ≤ CS.CPL else GPerror(NEW_SEL);
    if DT(NEW_SEL).DPL = CS.CPL else GPerror(NEW_SEL);
    if DT(NEW_SEL).P=1 else NPerror(NEW_SEL);
    if (SP-4) ≥ SS.CACHE.LIMIT (* スタックに戻りアドレスを PUSH す
        る空間があるかどうか *)

        else SSerror(0);
    if NEW_IP ≤ DT(NEW_SEL).LIMIT else GPerror(0);
    CS.CACHE ← DT(NEW_SEL);
    CS ← NEW_SEL;
    CS.RPL ← SS.CPL;
    IP ← NEW_IP;|
if DT(NEW_SEL) = CALL_GATE
then | (* NEW_SEL が指定するディスクリプタがコールゲートである場合 *)
    if CALL_GATE.DPL ≥ CS.CPL else GPerror(NEW_SEL);
    if CALL_GATE.DPL ≥ NEW_SEL.RPL
        else GPerror(NEW_SEL);
    if CALL_GATE.P = 1 else NPerror(NEW_SEL);
    if CALL_GATE.SEL ≠ ヌルセクタ else GPerror(0);
    if (CALL_GATE.SEL.INDEX+1)*8-1 ≤ DTR.LIMIT
        else GPerror(CALL_GATE.SEL);
    if DT(CALL_GATE.SEL).E=1
        else GPerror(CALL_GATE.SEL);
    if DT(CALL_GATE.SEL).DPL ≤ CS.CPL
        else GPerror(CALL_GATE.SEL);
    if DT(CALL_GATE.SEL).C=0 AND DT(CALL_GATE.SEL).DPL
        < CS.CPL
    then | (* より高い(数値的により小さい)特権レベル n のコードセグメントに制御を
        移行する場合 *)
        if TR.TSS.SSn ≠ ヌルセクタ (* TR.TSS.SSn は TR によ
            って指定される現在の TSS
            に記録された、特権レベル n
            で使用されるスタックセグメ
            ントのセグメントセクタを
            表す *)

            else TSerror(0);
        if (TR.TSS.SSn.INDEX+1)*8-1 ≤ DTR.LIMIT
            else TSerror(TR.TSS.SSn);

```

```

if TR.TSS.SSn.RPL = DT(CALL_GATE.SEL).DPL
  else TSError(TR.TSS.SSn);
if DT(TR.TSS.SSn).DPL = DT(CALL_GATE.SEL).DPL
  else TSError(TR.TSS.SSn);
if DT(TR.TSS.SSn).E=0 AND DT(TR.TSS.SSn).W=1
  else TSError(TR.TSS.SSn);
if DT(TR.TSS.SSn).P=1
  else SSError(TR.TSS.SSn);
if TR.TSS.SPn-CALL_GATE.WC*2-8
  ≥ DT(TR.TSS.SSn).LIMIT+1
  (* 特権レベル n で使用するスタックに、似前のスタックポインタ、戻り
    アドレス、そしてコールゲートに指定されたワードのパラメータを記
    録する領域があるかどうか *)
  else SSError(0);
if CALL_GATE.IP ≤ DT(CALL_GATE.SEL).LIMIT
  else GPErrror(0);
SS ← TR.TSS.SSn;
SP ← TR.TSS.SPn;
CS ← CALL_GATE.SEL;
IP ← CALL_GATE.IP;
CS.CACHE ← DT(CALL_GATE.SEL);
SS.CACHE ← DT(TR.TSS.SSn);
PUSH oldSS; (* 新しいスタックに似前のスタックの SS, SP を PUSH
              する *)
PUSH oldSP;
(* count ワードだけ、似前のスタックから新しいスタックにパラメータを
   コピーする *)
count ← CALL_GATE.WC AND 011111B;
while(count≠0)
  |PUSH oldSS:[oldSP+(count-1)*2];
  count ← count-1;|
PUSH 戻り CS;
PUSH 戻り IP;
SS.CPL ← DT(SS).DPL;
CS.RPL ← SS.CPL;|
else | (* 特権レベルが変化しない場合 *)
  if SP-4 ≥ SS.CACHE.LIMIT+1 else SSError(0);
  if CALL_GATE.IP ≤ DT(CALL_GATE.SEL).LIMIT
    else GPErrror(0);
  CS ← CALL_GATE.SEL;
  IP ← CALL_GATE.IP;
  PUSH 戻り CS;
  PUSH 戻り IP;
  CS.CACHE ← DT(CALL_GATE.SEL);
  CS.RPL ← SS.CPL;|

```

```

if DT(NEW_SEL) = TASK_GATE
then | (* NEW_SEL が指定するディスクリプタがタスクゲートである場合 *)
    if TASK_GATE.DPL ≥ CS.CPL else GPerror(NEW_SEL);
    if TASK_GATE.DPL ≥ NEW_SEL.RPL
        else GPerror(NEW_SEL);
    if TASK_GATE.P = 1 else NPerror(NEW_SEL);
    if TASK_GATE.SEL.TI = 0 (* TSS ディスクリプタは GDT に定義しなければならぬ *)
        else GPerror(TASK_GATE.SEL);
    if (TASK_GATE.SEL.INDEX+1)*8-1 ≤ GDTR.LIMIT
        else GPerror(TASK_GATE.SEL);
    if DT(TASK_GATE.SEL).B = 0
        else GPerror(TASK_GATE.SEL);
    if DT(TASK_GATE.SEL).P = 1
        else NPerror(TASK_GATE.SEL);
    TASK_SWITCH; (* TSS セレクタが TASK_GATE.SEL で指定されるタスクに
                  スイッチする *)
    if IP ≤ CS.CACHE.LIMIT else GPerror(0);|
if DT(NEW_SEL) = TSS_DESCRIPTOR
then | (* NEW_SEL が指定するディスクリプタが TSS ディスクリプタである場合 *)
    if DT(NEW_SEL).DPL ≥ CS.CPL else GPerror(NEW_SEL);
    if DT(NEW_SEL).DPL ≥ NEW_SEL.RPL
        else GPerror(NEW_SEL);
    if DT(NEW_SEL).B = 0 else GPerror(NEW_SEL);
    if DT(NEW_SEL).P = 1 else NPerror(NEW_SEL);
    TASK_SWITCH; (* TSS セレクタが NEW_SEL で指定されるタスクにスイッチす
                  る *)
    if IP ≤ CS.CACHE.LIMIT else GPerror(0);|
else (* CALL 命令が参照するセレクタが上のいずれでもない場合 *)
    GPerror(NEW_SEL);

```

リスト(e) far RET 命令を実行するとき、80286 は次のような処理を実行する。

```

if (SP+2) < 0FFFFH else SSerror(0);
if SS:[SP+2].RPL ≥ CS.CPL else GPerror(SS:[SP+2]);
if SS:[SP+2].RPL = CS.CPL
then | (* 同じ特権レベルへのリターンの場合 *)
    if SS:[SP+2] ≠ nulセレクタ else GPerror(0);
    if (SS:[SP+2].INDEX+1)*8-1 ≤ DTR.LIMIT
        else GPerror(SS:[SP+2]);
    if DT(SS:[SP+2]).E = 1 else GPerror(SS:[SP+2]);
    if DT(SS:[SP+2]).C = 0
        then | (* ノンコンフォーミングコードセグメントの場合 *)
            if DT(SS:[SP+2]).DPL = CS.CPL

```

```

        else GError(SS:[SP+2]);|
if DT(SS:[SP+2]).C=1
  then|(* コンフォーミング(特権例外)コードセグメントの場合 *)
    if DT(SS:[SP+2]).DPL ≤ CS.CPL
      else GError(SS:[SP+2]);|
if DT(SS:[SP+2]).P=1 else NError(SS:[SP+2]);
if SP > SS.CACHE.LIMIT+1 else SError(0);
if SS:[SP] ≤ DT(SS:[SP+2]).LIMIT else GError(0);
CS ← SS:[SP+2];
IP ← SS:[SP];
CS.CACHE ← DT(SS:[SP+2]);
SP ← SP+4+imm;|(* imm は RET 命令のオペランドに指定した値を表すものと
                する *)
else|(* より低い特権レベルへリターンする場合 *)
  if SP+8+imm ≤ 0FFFFH (* 現在のスタックに戻りアドレスのポインタ, パラ
                        メータ, そして似前のスタックのポインタが記録
                        されているか *)

    else SError(0);
  if SS:[SP+2] ≠ ヌルセクタ else GError(0);
  if (SS:[SP+2].INDEX+1)*8-1 ≤ DTR.LIMIT
    else GError(SS:[SP+2]);
  if DT(SS:[SP+2]).E=1 else GError(SS:[SP+2]);
  if DT(SS:[SP+2]).C=0
    then|(* ノンコンフォーミングコードセグメントの場合 *)
      if DT(SS:[SP+2]).DPL = SS:[SP+2].RPL
        else GError(SS:[SP+2]);|
  if DT(SS:[SP+2]).C=1
    then|(* コンフォーミング(特権例外)コードセグメントの場合 *)
      if DT(SS:[SP+2]).DPL ≤ SS:[SP+2].RPL
        else GError(SS:[SP+2]);|
  if DT(SS:[SP+2]).P = 1 else NError(SS:[SP+2]);
  if SS:[SP+6+imm] ≠ ヌルセクタ
    (* SS:[SP+6+imm] には似前のスタックの SS のセクタが記録されてい
      る *)
    else GError(0);
  if (SS:[SP+6+imm].INDEX+1)*8-1 ≤ DTR.LIMIT
    else GError(SS:[SP+6+imm]);
  if SS:[SP+6+imm].RPL = SS:[SP+2].RPL
    else GError(SS:[SP+6+imm]);
  if DT(SS:[SP+6+imm]).E = 0
    AND DT(SS:[SP+6+imm]).W=1
    else GError(SS:[SP+6+imm]);
  if DT(SS:[SP+6+imm]).DPL = SS:[SP+2].RPL
    else GError(SS:[SP+6+imm]);
  if DT(SS:[SP+6+imm]).P = 1

```

```

    else NError(SS:[SP+6+imm]);
if SS:[SP] ≤ DT(SS:[SP+2]).LIMIT+1
    else GError(0);
SS.CPL ← SS:[SP+2].RPL;
CS ← SS:[SP+2];
IP ← SS:[SP];
CS.RPL ← SS.CPL;
SP ← SP+4+imm;
SS ← SS:[SP+2];
SP ← SS:[SP];
SP ← SP+imm;
CS.CACHE ← DT(CS);
SS.CACHE ← DT(SS);
(* DS, DS.CACHE に不正な値が入っていないか検査する. もしリターンした後の状
  況に合わないような値をもつときは DS にヌルセクタを代入し, DS.CACHE を使
  用できないようにする *)
if (DS.INDEX+1)*8-1 ≤ DTR.LIMIT
    else |DS ← 0; (* DS にヌルセクタ 0 を代入し, DS.CACHE を使用でき
                  ないようにする *)
        DS.CACHE ← invalid;|
if DT(DS).E=0 OR (DT(DS).E=1 AND DT(DS).R=1)
    else |DS ← 0; DS.CACHE ← invalid;|
if DT(DS).E=0 OR (DT(DS).E=1 AND DT(DS).C=0)
    then |if DT(DS).DPL ≥ MAX(CS.CPL, DS.RPL)
        else |DS ← 0; DS.CACHE ← invalid;|
(* ES についても, DS の場合と全く同様の処理を実行する *)

```

リスト(f) 割り込みタイプ INT\_VEC の割り込みにおいて, 80286 は次の処理を実行する.

```

if (INT_VEC+1)*8-1 ≤ IDTR.LIMIT
    else GError(INT_VEC*8+2+EXT)
if (IDT(INT_VEC) = INT_GATE) OR (IDT(INT_VEC) = TRAP_GATE
    OR (IDT(INT_VEC) = TASK_GATE
    (* INT_VEC の指定するゲートが割り込みゲート, トラップゲート, タスクゲートのいずれ
      かである *)
    else GError(INT_VEC*8+2+EXT);
if INTinstruction(* INT 命令などのソフトウェア割り込みによって割り込み処理を実
                  行する場合 *)
    then |if IDT(INT_VEC).DPL ≥ CS.CPL
        else GError(INT_VEC*8+2+EXT);|
if IDT(INT_VEC).P=1 else NError(INT_VEC*8+2+EXT);
if (IDT(INT_VEC) = INT_GATE) OR (IDT(INT_VEC) = TRAP_GATE)

```

```

then! (* 割り込みゲートまたはトラップゲートの場合 *)
  if IDT(INT_VEC).SEL ≠ ヌルセクタ else GError(EXT);
  if (IDT(INT_VEC).SEL.INDEX+1)*8-1 ≤ DTR.LIMIT
    else GError(IDT(INT_VEC).SEL+EXT);
  if DT(IDT(INT_VEC).SEL).E = 1
    else GError(IDT(INT_VEC).SEL+EXT);
  if DT(INT_VEC).SEL).P = 1
    else NError(IDT(INT_VEC).SEL+EXT);
  if DT(IDT(INT_VEC).SEL).C = 0
    AND (DT(IDT(INT_VEC).SEL).DPL < CS.CPL)
  then! (* ノンコンフォーミングコードセグメントで、より高い特権レベルに移行する
    場合 *)
    if TR.TSS.SSn ≠ ヌルセクタ else GError(EXT);
    if (TR.TSS.SSn.INDEX+1)*8-1 < DTR.LIMIT
      else TError(SSn+EXT);
    if TR.TSS.SSn.RPL = DT(IDT(INT_VEC).SEL).DPL
      else TError(SSn+EXT);
    if DT(TR.TSS.SSn).DPL = DT(IDT(INT_VEC).SEL).
      DPL
      else TError(SSn+EXT);
    if (DT(TR.TSS.SSn).E=0)
      AND (DT(TR.TSS.SSn).W=1)
      else TError(SSn+EXT);
    if DT(TR.TSS.SSn).P = 1
      else SError(SSn+EXT);
    if TR.TSS.SPn-10 ≥ DT(TR.TSS.SSn).LIMIT+1
      else SError(0);
    if IDT(INT_VEC).IP ≤
      DT(IDT(INT_VEC).SEL).LIMIT
      else GError(0);
    SS ← TR.TSS.SSn;
    SP ← TR.TSS.SPn;
    CS ← IDT(INT_VEC).SEL;
    IP ← IDT(INT_VEC).IP;
    CS.CACHE ← DT(CS);
    SS.CACHE ← DT(SS);
    PUSH oldSS;
    PUSH oldSP;
    PUSH flag register;
    PUSH 戻り CS;
    PUSH 戻り IP;
    SS.CPL ← DT(IDT(INT_VEC).SEL).DPL;
    CS.RPL ← SS.CPL
    if error_code then(* エラーコードがある場合 *)
      PUSH error_code;

```

```

if IDT(INT_VEC) = INT_GATE
  then (* 割り込みゲートの場合 *) FLAG.IF ← 0;
FLAG.TF ← 0;
FLAG.NT ← 0;|
if (DT(IDT(INT_VEC).SEL).C=1) OR (DT(IDT(INT_VEC).
  SEL).DPL = CS.CPL)
  then|(* コンフォーミングコードセグメントか特権レベルが変化しない場合
    *)
    if (SP-6) ≥ SS.CACHE.LIMIT+1
      else SSError(0);
    if error_code
      then|(* エラーコードがある場合 *)
        if (SP-8) ≥ SS.CACHE.LIMIT+1
          else SSError(0);|
    if IDT(INT_VEC).IP ≤
      DT(IDT(INT_VEC).SEL).LIMIT
      else GPErrror(0);
    PUSH flag register;
    PUSH 戻り CS;
    PUSH 戻り IP;
    CS ← IDT(INT_VEC).SEL;
    IP ← IDT(INT_VEC).IP;
    CS.CACHE ← DT(IDT(INT_VEC).SEL);
    CS.RPL ← SS.CPL;
    if error_code then (* エラーコードがある場合 *)
      PUSH error_code;
    if IDT(INT_VEC) = INT_GATE
      then (* 割り込みゲートの場合 *) FLAG.IF ← 0;
    FLAG.TF ← 0;
    FLAG.NP ← 0;|
  else GPErrror(IDT(INT_VEC).SEL+EXT);
if IDT(INT_VEC) = TASK_GATE
  then|(* 割り込みによってタスクゲートを参照した場合 *)
    if TASK_GATE.SEL.TI = 0
      (* TSS ディスクリプタは GDT に定義しなければならない *)
    else GPErrror(TASK_GATE.SEL);
    if (TASK_GATE.SEL.INDEX+1)*8-1 ≤ GDTR.LIMIT
      else GPErrror(TASK_GATE.SEL);
    if DT(TASK_GATE.SEL).B = 0
      else GPErrror(TASK_GATE.SEL);
    if DT(TASK_GATE.SEL).P = 1
      else NPerrror(TASK_GATE.SEL);
    TASK_SWITCH;(* TSS セレクタが TASK_GATE.SEL で指定され
      る新しいタスクにスイッチする *)
    if error_code

```

```

    then | (* エラーコードがある場合 *)
        if SP-2 ≥ SS.CACHE.LIMIT+1
            else SSerror(0);
        PUSH error_code;|
    if IP ≤ CS.CACHE.LIMIT else GPerror(0);|

```

リスト(g) IRET 命令において、80286 は次の処理を実行する

```

if FLAG.NT = 1
    then | (* タスクスイッチとして動作する *)
        if TR.TSS.BACK_LINK.TI = 0
            else TSerror(TR.TSS.BACK_LINK);
        if (TR.TSS.BACK_LINK.INDEX+1)*8-1 ≤ GDTR.LIMIT
            else TSerror(TS.TSS.BACK_LINK);
        if DT(TR.TSS.BACK_LINK).AR = TSS_DESCRIPTOR
            (* TSS のバックリンクに記録されたセクタが TSS ディスクリプタを指定する *)
            else TSerror(TR.TSS.BACK_LINK);
        if DT(TR.TSS.BACK_LINK).B=1
            else TSerror(TR.TSS.BACK_LINK);
        if DT(TR.TSS.BACK_LINK).P=1
            else TSerror(TR.TSS.BACK_LINK);
        TASK_SWITCH; (* BACK_LINK で指定される TSS セクタの新しいタスクに
                       スイッチする *)
        if IP ≤ CS.CACHE.LIMIT else GPerror(0);|
if FLAG.NT = 0
    then | (* 割り込みからのリターン命令として動作する *)
        if (SP+2) < 0FFFFH else SSerror(0);
        if SS:[SP+2].RPL ≥ CS.CPL
            else GPerror(SS:[SP+2]);
        if SS:[SP+2].RPL = CS.CPL
            then | (* 同じ特権レベルのセグメントに制御を移行する場合 *)
                if (SP+4) < 0FFFFH else SSerror(0);
                if SS:[SP+2] ≠ ヌルセクタ else GPerror(0);
                if (SS:[SP+2].INDEX+1)*8-1 ≤ DTR.LIMIT
                    else GPerror(SS:[SP+2]);
                if DT(SS:[SP+2]).E = 1
                    else GPerror(SS:[SP+2]);
                if DT(SS:[SP+2]).C = 0
                    then | (* ノンコンフォーミングコードセグメントの場合 *)
                        if DT(SS:[SP+2]).DPL = CS.CPL
                            else GPerror(SS:[SP+2]);|
                    else | (* コンフォーミングコードセグメントの場合 *)
                        if DT(SS:[SP+2]).DPL ≤ CS.CPL

```

```

        else GPerror(SS:[SP+2]);|
if DT(SS:[SP+2]).P=1
    else NPerror(SS:[SP+2]);
if SS:[SP] ≤ DT(SS:[SP+2]).LIMIT
    else GPerror(0);
CS ← SS:[SP+2];
IP ← SS:[SP];
CS.CACHE ← DT(CS);
FLAG ← SS:[SP+4];
SP ← SP+6;|
else|(* より低い特権レベルにリターンする場合 *)
    if (SP+8) < OFFFFH else SSerror(0);
    if SS:[SP+2] ≠ ヌルセクタ
        else GPerror(0);
    if (SS:[SP+2].INDEX+1)*8-1 ≤ DTR.LIMIT
        else GPerror(SS:[SP+2]);
    if DT(SS:[SP+2]).E = 1
        else GPerror(SS:[SP+2]);
    if DT(SS:[SP+2]).C = 0
        then|(* ノンコンフォーミングコードセグメントの場合 *)
        if DT(SS:[SP+2]).DPL = CS.CPL
            else GPerror(SS:[SP+2]);|
        else|(* コンフォーミング (特権例外) コードセグメントの場合 *)
            if DT(SS:[SP+2]).DPL > CS.CPL
                else GPerror(SS:[SP+2]);|
    if DT(SS:[SP+2]).P = 1
        else NPerror(SS:[SP+2]);
    if SS:[SP+8] ≠ ヌルセクタ else GPerror(0);
    if (SS:[SP+8].INDEX+1)*8-1 ≤ DTR.LIMIT
        else GPerror(SS:[SP+8]);
    if SS:[SP+8].RPL = SS:[SP+2].RPL
        else GPerror(SS:[SP+8]);
    if DT(SS:[SP+8]).E = 0
        AND DT(SS:[SP+8]).W = 1
        else GPerror(SS:[SP+8]);
    if DT(SS:[SP+8]).DPL = SS:[SP+2].RPL
        else GPerror(SS:[SP+8]);
    if DT(SS:[SP+8]).P = 1
        else NPerror(SS:[SP+8]);
    if SS:[SP] ≤ DT(SS:[SP+2]).LIMIT
        else GPerror(0);
CS ← SS:[SP+2];
IP ← SS:[SP];
FLAG ← SS:[SP+4];

```

```

SS ← SS:[SP+8];
SP ← SS:[SP+6];
CS.RPL ← SS:[SP+2].RPL;
CS.CACHE ← DT(CS);
SS.CACHE ← DT(SS);
(* DS, DS.CACHE に不正な値が入っていないか検査する. もし,
リターンした後の状況に合わないような値をもつときは DS
にヌルセクタを代入し, DS.CACHE を使用できないように
する *)
if (DS.INDEX+1)*8-1 ≤ DTR.LIMIT
  else | DS ← 0; DS.CACHE ← invalid; |
if DT(DS).E = 0 OR (DT(DS).E = 1
  AND DT(DS).R = 1)
  else | DS ← 0; DS.CACHE ← invalid; |
if DT(DS).E = 0 OR (DT(DS).E = 1
  AND DT(DS).C = 0)
  then | if DT(DS).DPL ≥ MAX(CS.CPL, DS.RPL)
        else | DS ← 0; DS.CACHE ← invalid; ||
(* ES についても DS の場合と全く同様の処理を実行する *)

```

リスト(h) TASK\_SWITCH TSS セクタ NEWTSS\_SEL で指定される新しいタスクにスイッチするとき 80286 は次のような手順を実行する.

```

DT(NEWTSS_SEL).B ← 1;
if TR.CACHE.LIMIT ≥ 43 else TSError(NEWTSS_SEL);
if DT(NEWTSS_SEL).LIMIT ≥ 43 else TSError(NEWTSS_SEL);
TR.TSS ← (AX, BX, CX, DX, SI, DI, BP, SP, IP, FLAG, CS, DS, ES, SS,
  LDTR);
(* 現在のレジスタの状態を現在の TSS に記録する *)
if CALL or interrupt cause task switch
  then (* CALL 命令または割り込みによって実行されるタスクスイッチの場合 *)
    NEWTSS_SEL.TSS.BACK_LINK ← TR; (* 新しいタスクの TSS のバック
    リンクに現在のタスクの TSS セ
    レクタを記録する *)

DT(TR).B ← 0;
TR ← NEWTSS_SEL;
TR.CACHE ← DT(TR);
(AX, BX, CX, DX, SI, DI, BP, SP, IP, FLAG, CS, DS, ES, SS, LDTR)
← TR.TSS;
(* 新しい TSS に記録されていたレジスタの状態を 80286 のレジスタに代入する *)
(* LDTR キャッシュとセグメントキャッシュは新しい値を代入するまで使用できないようにす
る *)
LDTR.CACHE ← invalid;
SS.CACHE ← invalid;

```

```

CS.CACHE ← invalid;
DS.CACHE ← invalid;
ES.CACHE ← invalid;
MSW.TS ← 1;
if CALL or interrupt cause task switch
  then (* CALL 命令, 割り込みによって実行されるタスクスイッチの場合 *)
    FLAG.NT ← 1;
if (TR.TSS.LDT_SEL.INDEX+1)*8-1 ≤ GDTR.LIMIT
  (* ここで, TR.TSS.LDT_SEL は現在の TSS に記録された LDT セクタを表す *)
  else TSError(TR.TSS.LDT_SEL);
if DT(TR.TSS.LDT_SEL) = LDT_DESCRIPTOR
  else TSError(TR.TSS.LDT_SEL);
if DT(TR.TSS.LDT_SEL).P = 1 else TSError(TR.TSS.LDT_SEL);
LDTR.CACHE ← DT(LDTR);
CS.CPL ← TR.TSS.CS.RPL
if SS ≠ ヌルセクタ else TSError(SS);
if (SS.INDEX+1)*8-1 ≤ DTR.LIMIT else TSError(SS);
if SS.RPL = CS.CPL else TSError(SS);
if DT(SS).DPL = CS.CPL else TSError(SS);
if DT(SS).E = 0 AND DT(SS).W = 1 else TSError(SS);
if DT(SS).P = 1 else TSError(SS);
SS.CACHE ← DT(SS);
if CS ≠ ヌルセクタ else TSError(CS);
if (CS.INDEX+1)*8-1 ≤ DTR.LIMIT else TSError(CS);
if DT(CS).E = 1 else TSError(CS);
if DT(CS).C = 0
  then (* ノンコンフォーミングコードセグメントの場合 *)
    if DT(CS).DPL = CS.CPL else TSError(CS);|
  else (* コンフォーミングコードセグメントの場合 *)
    if DT(CS).DPL ≤ CS.CPL else TSError(CS);|
if DT(CS).P = 1 else NPErrror(CS);
CS.CACHE ← DT(CS);
(* DS キャッシュの決定 *)
if DS ≠ ヌルセクタ
  then if (DS.INDEX+1)*8-1 ≤ DTR.LIMIT else TSError(DS);
        if DT(DS).E = 0 OR (DT(DS).E = 1 AND DT(DS).R=1)
          else TSError(DS);
        if DT(DS).E = 0 OR (DT(DS).E = 1 AND DT(DS).R = 1
          AND DT(ES).C = 0)
          then if DT(DS).DPL ≥ CS.CPL else TSError(DS);
                if DT(DS).DPL ≥ DS.RPL else TSError(DS);|
        if DT(DS).P = 1 else NPErrror(DS);
        DS.CACHE ← DT(DS);|
(* ES キャッシュの場合も上の DS キャッシュとまったく同様に決まる *)

```

## 参 考 文 献

- (1) iAPX 286 プログラマーズリファレンスマニュアル
- (2) iAPX 286 ハードウェアリファレンスマニュアル
- (3) iAPX 286 オペレーティングシステムライターズガイド
- (4) iSBC 286/10 A single board computer hardware reference manual

## お わ り に

インテル社製品のよいところはマイクロプロセッサのチップとともに、すぐれた開発環境と iRMX, XENIX などのようなソフトウェア環境が提供されるところにもある。逆にいえば、インテル社の提供するアセンブリ言語、PL/M 言語などの高級言語、各種のユーティリティソフト、そして iRMX などの OS などを含めてのソフトウェア環境を知らずに、チップのアーキテクチャだけを勉強した場合、そのマイクロプロセッサを誤解する可能性がある。

たとえば、8086、80286 はセグメンテーションを採用しているからプログラムが困難であるという批判を耳にすることがあるが、それはおそらくセグメントを扱う方法が悪いために、プログラミングにおいて問題が発生しているのだろう。インテルのアセンブリ言語 (ASM 86, ASM 286 など) がもついくつかの疑似命令は、セグメントをもつマイクロプロセッサのプログラミング上の困難さを解決する。なお、セグメント自体はミニコン以上のレベルの汎用計算機のアーキテクチャとして提案され、使用されてきた考え方であり、もし 8086 のセグメントを批判するとすれば、それはセグメント自体にではなく、セグメントの最大の大きさが 64 K バイトの制限をもつことに対してであろう。しかし、このことも 8086 が開発された時代は、そのときのメモリチップの容量を考えれば、グラフィックワークステーションなどのように大量のリニアなメモリ空間を必要とする応用にマイクロプロセッサを利用することは考えられなかった時期であり、無理のないことである。80286 は 8086 のソフトウェアをマルチタスク環境で実行するということが第一義にあるから、セグメントの最大の大きさは 64 K バイトであるが、80386 のセグメントの最大の大きさは 4 G バイトになっている。また、80286 の GDT, LDT, IDT, またそれらのテーブルで定義するディスクリプタ、そして TSS を扱う困難さはビルダ BLD 286 が解決する。



# 索引

## ア 行

アクセスライト	60
アクセスライトフィールド	46
アドレスバス	5
一般保護エラー	139
インサーキットエミュレータ	192
インタフェース	5
インデックスレジスタ	22
ウェイトサイクル	10
エクスポートファイル	191
エラーコード	103, 126, 134
オフセット	13
オープンコレクタ	150
オペレーティングシステム	4
親プロセス	124

## カ 行

開発システム	188
加 算	27
仮想アドレス	62
カーネル	72
関数命令	176
間接 CALL	32

間接指定	22
間接 JMP	32
奇数バンク	155
逆演算命令	176
共有メモリ	164
偶数バンク	155
クロックジェネレータ 82284	144
グローバルディスクリプタテーブル	51
現在の特権レベル	75
減 算	27
コードフェッチ	13
コプロセッサ	165
子プロセス	124
コマンドサイクル	8
コマンド信号	5
コールゲート	81
コントロールレジスタ	168
コンフォーミングコードセグメント	88

## サ 行

再実行	142
三角関数	177
算術演算	27
算術演算命令	175

指数関数	178
システムクロック	8, 144
システムバス	163
実メモリ	62
シフト	27
シフト演算	28
時分割	4, 64
シミュレータ	188
シャットダウン状態	140
条件 JMP	32
乗算	28
除算	28
シングルタスク	112
スタックエラー	136
スタックセグメント	84
スタックフレーム	40
スタックマシン	166
ステータスサイクル	8
ステータス信号	150
ステータスレジスタ	168
ストリング命令	35
制御バス	5
静的システム	190
静的な変数	40
セグメンテーション	13
セグメント	13
セグメントオーバーライドインスト ラクションプリフィックス	15, 38
セグメントキャッシュ	11, 18, 46, 60
セグメントセクタ	13, 46
セグメントディスクリプタ	46, 49
セグメントレジスタ	11, 46
相対アドレス	13
ソフトウェア割り込み	98, 107

タ 行

対数関数	178
タイムシェアリングシステム	113, 114
タイムスライシング	4
タグワード	168
タスク	113
タスク管理	114
タスクゲート	125
タスクスイッチ	121, 184
ダブルワード	22
チップセレクト端子	161
直接指定	22
定数定義命令	178
ディスクリプタテーブル	65
ディスパッチャ	127
ディスプレイメント	22
デスティネーション	24
データバス	5
データベースシステム	72
手続き	40
デバイスドライバ	72
テーブルインジケータ	51
テンポラリリアル	169
電力供給	144
動的システム	190
動的な変数	40
特別なストローブ信号	156
特権	72
特権保護	78
特権命令	74
特権レベル	72, 74
トラップゲート	100

トロイの木馬.....93

## ナ 行

内部割り込み.....98, 110

二重エラー.....140

286 モード.....17

ニュークリアス.....72

ヌルセクタ.....55, 87

ヌルディスクリプタ.....66

## ハ 行

バイト.....22

パイプラインアクセス.....159

パイプラインドアドレッシング.....8

バインダ.....189

バス.....5

バスアービタ.....164

バスコントローラ 82288.....153

バスサイクル.....8

バスマスタ.....162

8259 A.....98, 105

82289.....164

80386.....51

80286 と 80287 の接続.....179

86 モード.....17

バックリンクセクタ.....122

バックワードリンク.....116

ハードウェア割り込み.....98

汎用レジスタ.....11

標準バス.....163

ビルダ.....189

ビルダプログラム.....189

ピングリッドアレイパッケージ.....7

物理アドレス.....13, 18

プロセッサクロック.....8, 144

プロテクティドバーチャルアドレス

モード.....17

プロテクトモード.....17

ベースアドレス.....46

ベースアドレスフィールド.....18

ベースレジスタ.....22

ポインタ.....16

保護例外.....110, 134

ホルト状態.....140

## マ 行

マシンステータスワード.....11

マルチタスク.....2, 113

マルチユーザシステム.....64

メモリ.....5

メモリ空間.....13

メモリスワップ.....62

メモリマップト I/O.....161

メモリライトコマンド.....153

メモリリードコマンド.....153

## ラ 行

ライト.....5

リアルアドレスモード.....13, 17

リアルタイムシステム.....2, 64, 113

リアルモード.....17

リード.....5

リードレスチップキャリアパッケージ.....7

リミット.....60

リミットフィールド.....18, 46

例外的な特権保護	88
例外割り込み	47
レジスタスタック	166, 171
ローカルディスクリプタテーブル	51
ローカルバス	162
ローテイト	27, 28
論理アドレス	13, 18, 46, 62
論理演算	27

## ワ 行

割り込み	125
割り込みアクノリッジサイクル	98, 105
割り込みアクノリッジ信号	153
割り込みゲート	100
割り込み処理	100
ワード	22
ワードカウント	81

## アルファベット

AR フィールド	19, 46
ARPL 命令	93
BLD 286	189
BND 286	189
BOUND 命令	108
BUSY	181
CALL 命令	30
CKM 端子	180
CLK	8
CLK 端子	144
CLTS 命令	186
CPL	75
CPU	5
CS 端子	161

CS の保護	58
DMA コントローラ	162
DPL	74
DS, ES の保護	55
EM	184
ENTER 命令	40
ERROR 端子	183
ESC 命令	181
far CALL 命令	30, 79, 81, 121, 125
far JMP 命令	30, 79, 81, 121, 125
far RET 命令	30
FLAG	11
FLD 命令	173
FNCLEX 命令	183
FST 命令	173
GDT	51
GDTR	51
HLDA 信号	162
HOLD 信号	162
ICE	192
IDT	100
IDTR	100
I/O	5
INT 3 命令	108
INTO 命令	108
INTR 端子	105
I/O インタフェース	160
I/O 空間	96
I/O ライトコマンド	153
I/O リードコマンド	153

IOPL .....	11, 96
IRET 命令 .....	104, 121
JMP 命令 .....	30
LDT .....	51, 117
LDT ディスクリプタ .....	54
LDTR .....	53, 118
LDTR キャッシュ .....	53, 118
LEAVE 命令 .....	40, 44
LGDT 命令 .....	53
LLDT 命令 .....	54, 118
LMSW 命令 .....	17
LOOP 命令 .....	32
LTR 命令 .....	116
MOV 命令 .....	23
MP .....	184
MSW .....	11, 184, 186
near CALL .....	30
near JMP .....	30
near RET .....	30
NMI .....	105
NT .....	11
OS .....	4, 72
OS の保護 .....	72
P ビットエラー .....	138
PCLK .....	8
PEACK .....	181

PEREQ .....	181
POP .....	14, 25
POP 操作 .....	174
PUSH .....	14, 25
$\overline{\text{READY}}$ 信号 .....	147
REP プリフィックス .....	38
RESET 信号 .....	147
RET 命令 .....	30, 86
SGDT 命令 .....	53
short JMP .....	30
SLDT 命令 .....	54
SMSW 命令 .....	17
SS の保護 .....	57
ST .....	171
task status segment .....	84
TI .....	51
TR .....	116
TR キャッシュ .....	116
TS .....	184
TSS .....	84, 113
TSS エラー .....	137
TSS ディスクリプタ .....	114
UDI .....	188
WAIT 命令 .....	181
XCHG 命令 .....	23



## 著 者 略 歴

昭和 53 年 近畿大学理工学部電子  
工学科卒業  
インテルジャパン株式  
会社を経て  
現 在 日本カルコン株式会社

## 図解 16 ビットマイクロコンピュータ 8 0 2 8 6 の 使 い 方

© 本岡善剛 1987

1987 年 5 月 20 日 第 1 版第 1 刷発行  
1989 年 5 月 20 日 第 1 版第 4 刷発行

OHM・OHM・OHM・  
OHM・OHM・OHM・  
OHM・OHM・OHM・  
OHM・OHM・OHM・  
OHM・OHM・OHM・  
OHM・OHM・OHM・  
OHM・OHM・OHM・  
OHM・OHM・OHM・

著者承認  
検印省略

著 者 もと本 さか岡 よし善 たけ剛

発 行 者 株式会社 オ ー ム 社  
代 表 者 種 田 則 一

発 行 所 株式会社 オ ー ム 社  
郵便番号 101  
東京都千代田区神田錦町 3-1  
振 替 東京 6 - 2 0 0 1 8  
電 話 03(233)0641(代表)

Printed in Japan

印刷 中央印刷 製本 関川製本所  
落丁・乱丁本はお取替えいたします

ISBN 4 - 274 - 07349 - 1

マイクロコンピュータハンドブック	渡辺・正田共編 矢田	B 5 判
オンラインシステムの設計	甘利直幸著	A 5 判
プログラマ適性検査	三重野重三著	A 5 判
入門ソフトウェア	高橋守清著	A 5 判
JISに準拠した BASICによるFORTRAN (演習コース)	大泉充郎監修	A 5 判
システム指向プログラミング	永村文孝著	B 5 判
マイクロコンピュータ入門テキスト	湯田・伊藤共著	A 5 判
絵ときマイコン実習	伊藤・橋本共著	A 5 判
続制御用マイコンの作り方・使い方	北川一雄著	B 5 判
マイコンロボットの設計と製作	堀・菅谷共編	B 5 判
F M シリーズ機械語入門	脇英世著	A 5 判
Z-80 機械語によるプログラムと制御	中山章著	A 5 判
図解 16ビットマイクロ コンピュータ 80286 の使い方	本岡善剛著	A 5 判
図解 6800/6809 によるマイコンアセンブラ入門	桐山清著	B 5 判
図解マイクロ コンピュータ 8086 アセンブラプログラミング入門	井出裕巳著	B 5 判
絵ときコンカレント C P / M 入門	荒武達男編	B 5 判
IBM 5550 活用法	鈴木智彦編著	B 5 判
IBM 5550 と J X B A S I C 入門	鈴木昇編	B 5 判
FACOM 9450 シリーズ B A S I C 入門	鈴木昇編著	B 5 判
わかる C A D / C A M	ばんざい秀男著	B 5 判
パソコンネットワーク入門	松山佐和著	B 5 判
パソコンマーケティング入門	阿部将美著	B 5 判
パーソナルコン ピュータのための O S - 9 入門	蜂谷博著	A 5 判
パソコンデータベース活用法	酒井・葛井共著 阿部	B 5 判
パソコンによる パーソナルプロッタ・プログラミング	杉浦義人著	B 5 判
パソコンによる 透視図の作り方	岡本博著	B 5 判
家庭でパ ソコン入門	岡田千恵子著	B 5 判
親子でパ ソコン教室	岩佐澄男監修	B 5 判
小学生の ためのパ ソコン教室	松山佐和共著 松山優治	B 5 判
中学生の ためのパ ソコン教室	吉田・木村共著	B 5 判
中学生の ための B A S I C 教室	吉田・木村共著	B 5 判
高校生の ためのパ ソコン教室	竹内・目七共著	B 5 判





## 図解コンピュータシリーズ

江村潤朗 監修

### コンピュータ・システム入門

江村潤朗・野津 昭 共著 (A5・p.210)

### ハードウェア・システム入門

江村潤朗 著 (A5・p.216)

### FORTRAN 入門

海老沢成享 著 (A5・p.230)

### COBOL 入門

海老沢成享 著 (A5・p.350)

### PL/I 入門

光吉民恵 著 (A5・p.244)

### PASCAL プログラミング入門

三沢常男・市川隆男 共著 (A5・p.264)

### BASIC プログラミング入門

平山静夫 著 (A5・p.290)

### アセンブラプログラミング入門

瀬下孝之・前田忠彦 共著 (B5・p.380)

### APL 入門

金子章弘 著 (A5・p.236)

### 日本語 APL 入門

平尾隆行 著 (A5・p.236)

### 簡易照会言語入門

—関係データベースシステム—

平尾隆行 著 (A5・p.228)

### ストラクチャード・プログラミング入門

國友義久 著 (A5・p.206)

### オペレーティング・システム入門

江村潤朗 著 (A5・p.176)

### ファイル編成入門

山谷正己 著 (A5・p.184)

### データベース入門

穂鷹良介 著 (A5・p.228)

### 仮想記憶システム入門

山谷正己 著 (A5・p.140)

### データ通信システム入門

保坂岩男 原著 石坂充弘 著 (A5・p.270)

### EDP システム設計入門

南條 優 著 (A5・p.204)

### オフィスコンピュータ入門

魚田勝臣・富沢研三 共著 (A5・p.202)

### オフィスオートメーション入門

中村 茂 著 (A5・p.238)

### RPG プログラミング技法

野口正雄 著 (B5・p.232)

### プログラム流れ図の作成技法

江村潤朗・野津 昭 共著 (B5・p.358)

### 対話式計算システムの活用技法

平尾隆行 著 (B5・p.262)

### プログラム開発管理

國友義久 著 (B5・p.184)

### 多重仮想記憶オペレーティングシステム

鎌田 肇 著 (B5・p.228)

### コンピュータとアプリケーション

寺沢康夫 著 (B5・p.236)

### データベース データ通信プログラミング

平尾隆行 著 (B5・p.220)

ISBN4-274-07349-1 C3055 P2369E



定価 2369 円 (本体 2300 円)